Initialisation Problems in Feature Composition

Thesis

UNRESTRICTED

Initialisation Problems in Feature Composition

Armstrong Nhlabatsi
(B.Eng, MSc.)

A thesis submitted in partial fulfilment of the requirements for the degree of Doctor of Philosophy in Computer Science

Department of Computing
Faculty of Mathematics, Computing, and Technology
The Open University

2009

Submission date: 22 June 2006
Date of award: 8 June 2009

Abstract

Composing features that have inconsistent requirements may lead to feature interactions that violate requirements satisfied by each feature in isolation. These interactions manifest themselves as conflicts on shared resources. Arbitration is a common approach to resolving such conflicts that uses prioritisation to decide which feature has access to resources when there is a conflict. However, arbitration alone does not guarantee satisfaction of the requirement of the feature that eventually gains access to a resource. This is because arbitration does not take into account that the resource may be in a state that is inconsistent with that expected by the feature. We call this the initialisation problem.

In this thesis we propose an approach to addressing the initialisation problem which combines arbitration with contingencies. Contingency means having several specifications per feature satisfying the same requirement, depending on the current resource state. We illustrate and validate our approach by applying it to resolving conflicts between features in smart home and automotive domains. The validation shows that contingencies complement arbitration by enabling satisfaction of the requirement of the feature that eventually gains access to a shared resource, regardless of the current state of the resource.

The main contribution of this thesis is an approach to analysing initialisation concerns in feature composition. At the core of our approach is an explicit consideration of all possible states of a resource as potential initial states. Given each initial state we then derive corresponding specifications that would enable a feature to satisfy its requirement in those states. We show that our approach to initialisation problems is relevant to addressing the feature interaction problem by characterising some types of conflicts as initialisation concerns.

Acknowledgements

My journey to the research reported in this thesis has been intellectually enriching. For this benefit I owe gratitude to a number of people who have helped me through this experience. First and foremost, I thank my supervisors; Prof Bashar Nuseibeh and Dr Robin Laney, for their support throughout and for their constructive criticism which helped me focus on an interesting research problem. I am very grateful to Prof Michael Jackson for his valuable feedback on early ideas that eventually lead to this thesis.

I am thankful to my colleagues in the Computing Department at the Open University; Mohammed Salifu, Yijun Yu, Michel Wermelinger and Thein Than Tun, for the helpful discussions that enriched the ideas in the thesis. I also thank Kurt Schneider for introducing me to the Instrument Cluster specification and the engineers at DaimlerChrysler. I gratefully acknowledge the Open University for providing structured research training and the EPSRC for the financial support.

Finally, I am very grateful to my family and friends (especially Rosena Chiumia) for being supportive, encouraging, and understanding through this exciting and sometimes stressful adventure.

Author Declaration

I hereby declare that the material presented in this thesis, except where references are made to related work, is original.

Armstrong Nhlabatsi.

Table of Contents

Table of Figures

x

Chapter 1. Introduction

A common approach to managing the complexity of developing large software systems is to decompose their functionality into features. A *feature* is a set of logically-related *requirements* and their *specifications,* intended to deliver a particular behavioural effect [Turner *et al.* 1999; Zave 2001; Calder *et al.* 2003; Bredereke 2004]. Our use of the terms 'requirement' and 'specification' is taken from Zave and Jackson [Zave and Jackson 1997]. A requirement is a statement of what behaviour a system is expected to exhibit. A specification is a description of how the behaviour that would satisfy a requirement will be accomplished. The responsibility of developing individual features may be allocated to different development teams [Palmer and Felsing 2002].

1.1 The Feature Interaction Problem

The individually developed features are then composed, to create a feature-based application. However, the composition of features with inconsistent requirements may lead to *feature interactions* [Keck and Kuehn 1998; Calder *et al.* 2003] – a phenomenon where features interfere with each other's behaviour in the composition. Such interference often leads to the violation of the requirements each feature satisfied in isolation. Feature interactions manifest themselves as conflicting actions of features on a *shared resource* [Bisbal and Cheng 2004]. The *feature interaction problem* is how to *detect* and *resolve* feature interactions.

The feature interaction problem has been studied in depth in the telecommunications domain. This is evidenced in the conference proceedings [Calder and Magill 2000; Reiff-Marganiec and Ryan 2005] and special issue journals [Logrippo 1998; Akyildiz *et al.* 2000; Amyot and Logrippo 2004; Reiff-Marganiec and Ryan 2007] documenting research results on proposed approaches to addressing this problem. A common characteristic of feature interactions is that

they result from sharing of *context*. By context we mean the properties of the world (such as resources) that features need to satisfy their requirements. This suggests that feature interaction is a *context sharing problem*. For example, for reactive systems such as telecommunication switching [Keck and Kuehn 1998] and flight control [Cortellessa *et al.* 2000] software, the feature interaction problem involves conflicts on the shared control of devices. Left unresolved, such conflicts often lead to incorrect operation and reduction in reliability for these systems.

When a feature interaction is detected at design time, features can be redesigned to eliminate it [Stafford and Wallnau 2001; Blair and Turner 2005; Calder and Miller 2006]. However, managing feature interactions at runtime is more challenging for two reasons: (1) feature redesign may not be possible; (2) runtime conflicts often have to be resolved with minimal manual intervention and within relatively short time limits. On the other hand, resolving feature interactions at runtime has the advantage of dealing with actual rather than potential conflicts. Moreover, postponing resolution to runtime avoids over-restricting the requirements to be satisfied by the specification of each feature. Despite the benefits of runtime resolution, current research has not advanced enough to resolve a majority of known types of feature interaction [Cameron *et al.* 1993; Kolberg *et al.* 2003; Shehata *et al.* 2007b].

1.2 Arbitration as a Feature Interaction Resolution Technique

A common approach to resolving runtime conflicts is to introduce an *arbitrator* [Tsang and Magill 1998; Hay and Atlee 2000; Laney *et al.* 2007]. When two parties are in conflict, one way to resolve the dispute is to refer it to a third party, called an arbitrator. The arbitrator considers the arguments of both parties and makes a binding decision on how the dispute should be resolved. In legal terminology this conflict resolution approach is called *arbitration* [Bonn 1972]. The concept of arbitration has been applied in a similar way in resolving feature interactions where an arbitrator intercedes between competing features and the shared resource.

Examples of arbitrators include the Feature Interaction Manager (FIM) [Tsang and Magill 1998], Composition Controller(CC) [Laney *et al.* 2007], Modular Supervisory Control with Priorities (MSCP) [Chen *et al.* 1995; Wong *et al.* 2000], and Conflict-and-Violation Free (CVF) composition operator [Hay and Atlee 2000]. Actions issued by features have to be approved by the arbitrator before they can be passed on to the resource. Arbitration alone resolves only conflicts resulting from *non-deterministic* compositions by *prioritising* features. Non-determinism occurs when two or more features require a shared resource to engage in different behaviours simultaneously, when the resource can engage in only one of the behaviours at a time [Cameron and Velthuijsen 1993]. For example, in a smart home [Kolberg *et al.* 2003], a climate control feature may require a window to be opened, while a security feature requires the window to be closed.

Arbitration ensures that in the event of a conflict a higher-priority feature is given exclusive control of the shared resource. However, this conflict resolution technique does not guarantee that the requirement of the feature that eventually gains control of the resource will be satisfied. This is because its model of the shared resource may be inconsistent with the actual shared resource state, due to the state having been changed by another feature that used the resource previously. This may result in its requirement not being satisfied even if granted access to the resource. We call this the *initialisation problem*. In this thesis we address the initialisation problem in feature composition.

1.3 The Initialisation Problem

In illustrating the initialisation problem consider the composition of two security features (*burglary capture* and *burglar deterrence*) in a smart home [Kolberg *et al.* 2003] which share a Digital Versatile Disc Recorder (abbreviated DVD-R). A DVD-R is an optical disc recording device that records onto a writable DVD media. We will use this as our running example in the rest of the thesis. The dynamic behaviour of the DVR is as shown in the state

machine in Figure 1.1. A dynamic behavioural description maps event occurrences to state changes. Depending on the current state, the occurrence of an event may result in change of state. It is this change of state of a resource in response to the occurrence of an event that enables a feature to satisfy its requirement. For example, if the current state is *Stopped*, the occurrence of a *play* event results in transition to a *Playing* state. Behavioural descriptions help in reasoning about how events issued according to specifications result in state changes on the resource which would eventually lead to the satisfaction of a requirement.

Features interacting with the DVD-R issue events from the set *{play, record, stop, pause}* to satisfy their requirements. In response to the occurrence of an event, and depending on its current state, the DVD-R may be in one of the following states: *Playing, Recording, Stopped, Paused_Recording, Paused_Playing.*

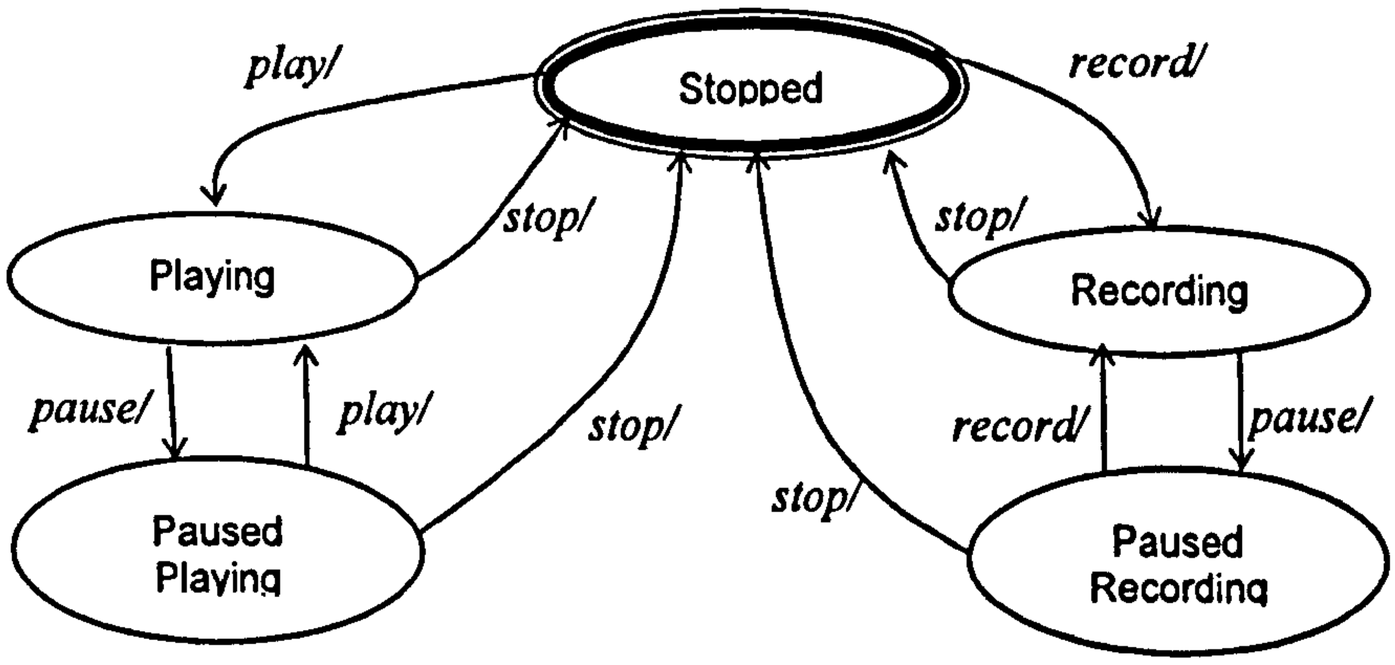


Figure 1.1 State Machine Description of DVD-R Behaviour

The burglary capture requirement is to record burglary footage from a security camera on the DVD-R. Its post-condition is that the DVD-R is in the *Recording* state. Meanwhile the burglar deterrence requirement is to play a movie on the DVD-R when the home owner is away to give the impression that someone is home. Similarly, its post-condition is that the DVD-R is in the *Playing* state. According to Figure 1.1, the burglary capture and burglar deterrence requirements can not be satisfied at the same time. This is because the DVD-R

cannot be playing and recording simultaneously. This is a non-deterministic conflict and is resolved with arbitration.

Specifications satisfy their requirements based on assumptions about the initial state of the resource. A specification that assumes a fixed state as its initial state may not always satisfy its requirement. This is because such a state can not be guaranteed to be true when the resource is shared. This is due to the possibility of other features changing the state. Such interference may lead to inconsistency between the actual state of the resource and that assumed by a feature about to engage with the resource.

For example, consider a scenario in which the burglar deterrence feature leaves the DVD-R in the *Playing* state. If the burglary capture feature assumes instead that the DVD-R is in the *Stopped* state then the capture requirement will not be satisfied if a burglar breaks-in. The burglar deterrence feature is said to have bypassed the burglary capture feature and this type of conflict is called a *bypass feature interaction* [Shehata *et al.* 2007b]. This conflict is due to the inability of the burglary capture feature to address initialisation concerns. Arbitration alone is insufficient in resolving this type of feature interaction. The above example demonstrates that initialisation is a significant problem as the initial state of the context determines whether a requirement will be satisfied or not.

1.4 Importance of Addressing Initialisation Problems in Feature Composition

When designing a software application that satisfies its requirements by changing the state of a physical resource (such as a device), it is often necessary to model the behaviour of the resource. The model acts a foundation on which reasoning about the behaviour and state of the resource is based. We use models in everyday life for documenting abstractions about the physical world. For example, a street map of a city is a useful tool for guiding visitors to places of interest in the city. Depending on the nature of the real-world phenomena being modelled, models can range from simple to complex. The street map example represents a

simpler model compared to models for complex systems such as those used in weather forecasting.

A model (normally) focuses on a particular aspect of the physical world, depending on the intended purpose. For example a city street map shows streets in relation to positions of landmarks. A meteorologist may be interested in a different kind of map – one showing air pressure patterns in the city. Similarly, a requirements analyst developing the specification of a scheduler to control a lift car in a building is interested in a model (such as a state machine) showing how the lift system responds to external control events. For example what events will make the lift start going up, stop at a floor, and open doors.

In this thesis we will be concerned about the latter types of models (those relevant to requirement analysts), that is, models of behaviour of resources that show events, states, and relations between event occurrence and state changes. We will focus on using these models in reasoning about conflicts instead of how they are created. We will assume that they have been created and they accurately capture the behaviour of a real-world resource.

Models about the behaviour of resources play an important role in the design of software systems since most reasoning about resource behaviour is based on the model. For this reason the correct operation of a control application and whether it eventually satisfies its requirements heavily relies on models of the resources being controlled. Models also have a profound implication for the initialisation problem in feature composition. This is because the genesis of initialisation problems is mismatch between state represented in a model and the actual state in the real world.

In this section we motivate the initialisation problem by making a case for why it is an important problem to address in requirements engineering. We explore possible consequences of ignoring initialisation concerns for two cases: (1) a feature executing in isolation and (2)

features in composition sharing resources. For each case we identify specific initialisation concerns and illustrate these problems with examples from real-life incidents.

1.4.1 Initialisation Concerns for Feature in Isolation

For a feature executing in isolation – having sole control of a resource – the initialisation problem involves addressing at least two concerns: *determining safe start-up state* and *model synchronisation*. Determining safe start-up state involves analysing the characteristics of both the world and the machine in order to determine when it is safe for the machine to engage with the world? For example before a lift scheduler can be executed for the first time it is important to consider at which floor the lift car should be initially so that it is safe to start interaction with it? Should it be in the ground floor or top floor? Having determined the initial floor position of the lift car, the installation technician should also initialise the model of the lift position in the scheduler accordingly.

Model synchronisation involves ensuring that the model is accurately synchronised with the actual current state of the real-world resource and continues to be synchronised for the rest of the execution time of a feature. For example for the scheduler to continue to serve lift service requests it is essential that its model is always correctly synchronised with the actual position of the lift car. At the least a mismatch between the position in the model and the actual lift position may result in the car stopping at the wrong floors. At worst, this may result in the lift car crashing to the ground floor!

1.4.2 Initialisation Concerns for Features in Composition

The initialisation problem for features in composition is more challenging than that of a feature executing in isolation because of the possibility of interference. In addition to the issues discussed in section 1.4.1, initialisation problems for features in composition involves

addressing the following concerns: *safe stoppage*, *safe start-up and resumption*, and *continuous model synchronisation*. We discuss these issues in detail below.

Safe Stoppage: At what state will it be safe to stop a feature currently using a resource so that another feature can take over the use of the resource? How might stopping a feature currently executing affect the satisfaction of requirements of machine that uses a shared resource later?

> In illustrating stoppage concerns consider the composition of the burglary capture feature and a *broadcast capture* feature such that the two features share the DVD-R whose behaviour is shown in figure 1.1. Assume that the broadcast capture requirement is to record a TV programme at a certain time and the burglary capture requirement is to record a burglary. In the following discussion assume that the burglary capture has a higher priority over broadcast capture. Consider a scenario in which the broadcast capture feature is set to record news from CNN between 7pm and 8pm. If a thief breaks into the house at 7:55pm, the recording of the news will have to stop to allow the burglary capture feature to capture a record of the burglar activity. Terminating the execution of the broadcast capture feature involves analysing the consequences of doing so on the satisfaction of both the broadcast and burglary requirements.

A first consideration is what state does the burglary capture feature expects the DVD-R to be before it can engage with it. Assume the burglary capture feature perceives the DVD-R to be in the *Stopped* state. Since there is a state mismatch we need to synchronise the model in the burglary capture feature with the actual state of the DVD-R. Can the DVD-R be forced to the desired Stopped state? What would be the consequences of doing so?

According to Figure 1.1, the DVD-R can be forced to the Stopped state by the occurrence of a *stop* event. However, abnormally aborting the news recording process in this way may result in a damaged DVD media. This additional concern arises from characteristics of the DVD media. By abnormal termination we mean stopping the recording without closing the session. Closing a recording session may take some time and this may give the thief enough time to get away. On the other hand, damaged DVD media can neither be read nor written on. Hence, the implication of a damage DVD is that both burglary capture and broadcast capture requirements may not be satisfied. Playback of the news recorded in the past 55 minutes may not be possible and the burglar activity may not be recorded.

A second consideration concerns how urgent is it that we should satisfy the burglary capture requirement. This involves answering the question: how important is the last five minutes of the news compared to the first five minutes of a burglary. Perhaps the last five minutes of the news is a summary and perhaps the first five minutes of burglary is the most useful. It may also be possible that the burglary capture feature may have been triggered but the burglar is not yet inside the house where he may be captured by the surveillance camera.

As the example illustrates, addressing stoppage issues in initialisation is a complex problem and involves a consideration of the characteristics of the shared resource.

Safe Start-up and Resumption: If a machine was suspended it is likely that the world could have been changed by another machine. How do we ensure that it resumes correctly? At which state can it be safe for the machine of a feature to engage with the world? How could the restarting/resumption of one machine affect machines of other features? We illustrate safe resumption issues with an incident reported in the Washington Post news website.

In March 2008 a nuclear power plant in Georgia (USA) was forced into an emergency shutdown for 48 hours[1] [Krebs 2008]. The plant has an automated safety system that shuts it down when a drop in water reservoirs that cool radioactive fuel rods is detected. An investigation into the cause of the incident revealed that it occurred after a software update was installed on a computer used for monitoring chemical and diagnostic data for one of the plant's control systems. The software update was meant to synchronise data on both the monitoring and safety systems. When the updated computer was rebooted, it reset the data on the control system. The automated safety systems incorrectly interpreted the lack of data to mean that cooling water reservoirs had drop below accepted levels - triggering an emergency shutdown of the plant as a result.

Both systems satisfied their requirements. The monitoring and control application initialised data on cooling water reservoirs level and the automated shutdown safety system triggered emergency shutdown when it detected unacceptable water levels. However, the combined behaviour of the two systems produced undesirable results. There are inherent design flaws in both systems which if addressed could have avoided this incident.

Firstly, the design of the monitoring and control system seems to have assumed that when it is initialised the cooling reservoirs would be empty hence the reset of the water level data value. In this instance this assumption was not true as the water levels were not initially empty when the monitoring application was started. More importantly, it appears the composition of the two subsystems did not consider how would starting the monitoring machine affect other machines already executing which share the reservoir water level data

Secondly, on the design of the automated safety system, it seems the implications of delegating the update of the water levels data monitoring to another application were not

[1] We thank Emmanuel Letier for bringing this example to our attention.

taken into account. If the safety system was directly reading the water levels, the possibility of reading a false value could have been minimised. Both of these design flaws may have been avoided had the engineers analysed initialisation concerns in the design of the systems in isolation and in their composition.

Continuous Model Synchronisation: When a feature is composed with other features, such that a resource is shared, there is a possibility that (with time) it may become out of sync with the true state of the resource even if it was initially in sync. This raises two closely related issues. (1) How to ensure that all features have a correct view of the current state of a resource they share? (2) If all features use a shared model of a resource state how should they be composed so as to safeguard against one feature changing the state in the shared model such that it is not consistent with the true state of the shared resource? We illustrate the consequences of a mismatch between the model and the real-world by the incident below.

Aeroperu Flight 603 was a scheduled flight from Peru to Chile which crashed in October 1996 [Anderson and Bambrick 2007][2]. The plane took-off at after midnight. A few seconds later, the cockpit emergency warning system was generating numerous alarms warning that the plane was flying too low. Contrary, the altimeter indicated that the plane had climbed to a safe altitude of approximately 9700 feet. The crew declared emergency and requested immediate return to the airport for emergency landing.

Initially they believed that the reading on the altimeter was the correct one. However, as the plane started to descend they discovered that the reading on the altimeter was not decreasing despite the drop in altitude. Making matters worse was that they had no visual reference since it was a night. Extremely confused about their true altitude, the crew requested help from air traffic control (ATC). ATC told them that they were flying at 7000 feet. It

[2] Broadcasted on National Geographic channel

turned out that both the ATC and altimeter information were wrong as they soon realised they were flying much lower. They attempted to climb but it was too late. The plane crashed into the ocean killing everyone onboard (9 crew members and 61 passengers).

An investigation revealed that the crash was caused by a piece of masking tape accidentally left covering static ports after cleaning the aircraft. Static ports are sensory devices for all flight instruments providing basic flight data such as airspeed and altitude to the pilots. As a result of the blocked static ports the altimeter relayed incorrect altitude. ATC also relayed incorrect altitude because the design was such that the information they had was calculated by onboard systems which relied on static ports. Consequently, the pilots did not know their true altitude and airspeed. Since it was at night, with no visual references, they could not navigate the plane – they were flying blind. This example illustrates the importance of having a model being always in correct synchrony with the real-world. The results of a mismatch can be tragic.

1.5 Using Contingencies to Address Initialisation

Contingencies complement arbitration by enabling satisfaction of the requirement regardless of the current state of the resource. Based on this observation, in this thesis we propose an approach to addressing the initialisation problem which ensures that in the event of a conflict the requirements of conflicting features are eventually satisfied. Our proposed approach combines arbitration with *contingency planning*.

Contingency planning is a concept from management science [Umanath 2003; Sousa and Voss 2008]. In management, contingency entails explicit a-priori statements about various situations which are not certain to happen but are nevertheless possible in the operations of an organisation. These situations are not part of the normal operations of the organisation and they are regarded as disruptions. Contingency planning is a risk management strategy aimed

at designing corresponding alternatives to how the satisfaction of organisational goals will be maintained should those situations arise.

In this thesis we use the term contingency to mean having several specifications per feature, satisfying the same requirement, depending on the current state of the shared resource. Our proposed approach involves deriving *contingent specifications* corresponding to each state of the shared resource at design time. Each specification satisfies the requirement of a feature given a particular state of the shared resource as an initial state. At run-time the specifications are composed through an arbitrator and selected for execution depending on the current state of the shared resource.

1.6 Thesis Contribution

Arbitration resolves conflicts by prioritising features contesting for a shared resource. However, arbitration alone is not sufficient as it does not guarantee that the requirement of the feature that eventually gains access to the shared resource will be satisfied – implying that the effort of applying arbitration could be futile. This is because arbitration does not address the initialisation problem. In order to ensure that the effort of applying arbitration is not wasted, it is therefore important that the initialisation problem is addressed.

The main contributions of this thesis are both conceptual and methodical. *Conceptually,* we propose an approach to analysing initialisation problems using the concept of contingency planning. We characterise bypass feature interactions as initialisation problems. This enables us to use our approach to addressing initialisation concerns to resolve bypass feature interactions. We then show that our approach to the initialisation problem can be combined with arbitration approaches. The result is a novel approach to feature interaction resolution which ensures that in the event of a non-deterministic conflict the requirement of a feature that is granted access to a resource is eventually satisfied.

Methodically, we present a method showing how our proposed conceptual approach can be applied in practical feature-driven software development using existing techniques and notations. We also provide a tool that automates the derivation of contingencies. Finally, we evaluate the proposed approach through its application to a case study of a practical problem.

1.7 Research Methodology

Our claim is that combining arbitration with contingencies aids the runtime resolution of non-determinism and bypass feature interactions. We substantiated this claim by applying the proposed approach to an example we have constructed and to a case study based on a practical problem. We created the laboratory constructed example such that it had characteristics that enabled us to evaluate essential attributes and demonstrate feasibility of our approach. However, although our example helped us to illustrate the concepts proposed in our approach in their simplest form, it is not representative of a real-life practical problem. As a result, the example was not enough to validate the practical relevance of our approach.

We used a case study to validate the practical relevance of our approach. Evaluation through the practical problem helped us in validating that the problem being solved by the proposed approach is a real problem, that is, it exists in real-life and it is not just a laboratory thought experiment. We found the practical case study very useful as a 'reality-check' as it allowed us to gauge the practical relevance our approach. It also revealed limitations of the approach which were otherwise not visible in the constructed example.

1.8 Thesis Structure

In Chapter 2 we present some background on the concept of 'feature' and how it fits in Requirements Engineering by exploring its relation to 'requirement' and 'specification'. Context is important in reasoning about feature interactions as conflicts manifest themselves on the context. We present the problem frames notation as a way of modelling features that makes context explicit. The derivation of specifications requires reasoning about the effects of

actions on the context that would bring about changes that satisfy the requirement. We introduce the Event Calculus as a language for reasoning about the effects of actions on context and automating the derivation of specifications.

Chapter 3 reviews approaches to detecting and resolving feature interactions with a focus on context sharing as a source of conflicts between features. We advance the argument that feature interaction is a context sharing problem by providing supporting evidence from the literature in the form of taxonomies and sources of feature interactions. Our review shows that the limitation of current approaches to feature interaction resolution is that they lack mechanisms for explicitly dealing with initialisation concerns and hence are insufficient in addressing conflicts resulting from the initialisation problem.

Our approach to resolving non-determinism and bypass feature interactions combines the concepts of arbitration and contingency planning. Chapter 4 presents the conceptual basis for our approach by motivating how the combination of the concepts of arbitration and contingencies are relevant to feature interaction resolution. Contingency planning enables features to deal with initialisation concerns. This is achieved by equipping each feature with contingent specifications corresponding to each state of the shared resource. Depending on the current state, one of the contingencies is selected to enable a feature to satisfy its requirement. Although contingencies may be sufficient in dealing with the initialisation problem, they are insufficient in resolving non-determinism as features may still conflict as they concurrently attempt to access a shared resource. In order to resolve non-determinism we argue that arbitration is necessary in feature composition to intercede between feature specifications and the shared resource. We show that arbitration is relevant to the resolution of non-determinism while contingencies resolve bypass interactions. We argue that a combination of the two concepts resolves both types of feature interactions.

Chapter 5 illustrates how the proposed approach can be used in practice by showing how an existing arbitration approach can be extended with contingencies. We present the steps involved in developing a feature-based application that makes use of the two concepts to resolve feature interactions. We identify two main steps such a development process could entail, namely: (1) building contingencies into specifications and (2) composing the contingent specifications through arbitration. The first step involves deriving contingent specifications. The derivation of contingent specifications can be erroneous and time-consuming if done manually. Chapter 6 presents a tool, called Contingency Specification Generator (CSG), which automates this task (derivation of contingent specifications).

In chapter 7 we report on an evaluation of the proposed approach through its application to a practical problem. Our evaluation shows that combining arbitration with contingencies ensures that in the event of a conflict the requirements of the features involved are eventually satisfied. Based on the evaluation we document limitations of the approach and possible alternatives to how they can be addressed. Finally, chapter 8 presents a summary of our work on feature specification and runtime composition, considers the application of the proposed approach to a much wider and general context, and suggests future directions for this research.

Chapter 2. Background

The concept of a 'feature' is central to the study of the feature interaction problem. However, in the feature interaction literature there is no generally agreed definition of what a feature is. Based on this observation, in section 2.1, we explore how this concept fits into Requirements Engineering by exploring its relation to the notion of a 'requirement'. Conflicts between features manifest themselves on shared resources. For this reason, we structure individual features and their compositions using the Problem Frames [Jackson 2001] approach to analysing and structuring software development problems. Problem Frames allow us to structure problems in a way that makes context and composition concerns explicit. We introduce the Problem Frames notation in section 2.2.

Section 2.3 presents the smart home feature interaction example introduced in section 1.3 in detail. We use this example in the rest of thesis to illustrate initialisation problems. A common approach to resolving conflicts on shared resources is arbitration. In this thesis we will use a Composition Controller [Laney *et al.* 2007] to illustrate arbitration and so a brief introduction to this approach is presented in section 2.4. Making shared context explicit is insufficient for reasoning about feature interactions because detection of conflicts requires knowledge about the dynamic behaviour of shared resources. In section 2.5 we introduce the Event Calculus [Shanahan 1999; Mueller 2006b] notation - a logic system that we use to express domain descriptions of shared resources.

2.1 Features and Requirements

Although the concept of a feature is commonly used in the feature interaction literature there is no generally agreed definition of what a feature is, beyond that it is '*additional, incremental, client-valued, and optional functionality*'. More importantly, with the exception

of [Laney *et al.* 2007; Classen *et al.* 2008], there is little evidence of work that explores how the concept of a feature relates to a *requirement* and *context* in Requirements Engineering. In this section we explore the different notions of a feature and propose a definition of a feature which is grounded on the entailment relation [Zave and Jackson 1997].

2.1.1 What is a feature?

Considering the functionality of a software system in terms of features is useful for two reasons. (1) It forms a common way of communicating user needs to application developers [Kang *et al.* 1998]. (2) It provides a means for grouping of system functionality which reduces complexity and eases maintenance [Zave and Jackson 2002; Bredereke 2004; Bredereke 2005]. However, in the feature interaction literature, there is no generally agreed definition of what is a feature. In this section we review the different definitions of a feature by considering the notion of a feature as an *optional or incremental unit of functionality, a client-valued function, and a functionality structuring concept*. We then propose a definition of a feature from an RE perspective.

Feature as Optional or Incremental Unit of Functionality: The feature interaction problem has been studied extensively in telecommunications. In this domain a feature is viewed as an optional or incremental unit of functionality [Keck and Kuehn 1998; Zave 2001; Calder *et al.* 2003], that provides additional functionality to an existing system [Braithwaite and Atlee 1994; Siddiqi and Atlee 2000b] - thereby extending the scope of its functionality [Fu *et al.* 2000]. The existing system consists of other features and the basic functionality of the application. This view of a feature stems from the fact that the basic functionality in a telephone switching system is essential (and necessary) to every feature that is added. This basic functionality satisfies the primary requirement of providing voice and data connections between caller and callee. Features such as *Call Forwarding*, provide variations of this basic

functionality by introducing additional (or incremental) constraints on call behaviour like re-routing a call to a different number when the called subscriber is busy.

Feature as a Client-Valued Function: Sochos *et al.* (2004) [Sochos *et al.* 2004] defines a feature as a client-valued function. This definition seems to be influenced by a product marketing perspective. In marketing the distinct characteristics of a product (those that make it stand out from competing products) are those mostly highlighted by a salesperson [Shaw *et al.* 1989]. This is consistent with the general definition of a feature as being a distinct characteristic of an object. Consumers of software products normally think of a software system in terms of the functionality that it offers. Based on this notion a feature is a user accessible or visible unit of functionality [Blair *et al.* 2002; Bisbal and Cheng 2004] or capability that is distinguishable and relevant to some stakeholders [Pang and Blair 2002; Pulvermueller *et al.* 2002].

Feature as a Functionality Structuring Concept: Turner *et al.* (1999) [Turner *et al.* 1999] argues that a feature should be considered as a functionality organising concept. Such a concept helps a system designer structure a software system into logically related chunks of functionality that makes system maintenance easier to comprehend. A similar view is also shared by Sochos *et al.* (2004) [Sochos *et al.* 2004] and Maccari and Heie (2005) [Maccari and Heie 2005] where a feature is defined as a logical unit of behaviour specified by a set of functional and quality requirements.

This notion considers a feature as a self-contained subset of system behaviour ; designed as a conceptual and cohesive chunk of functionality [Hall 2000a] ; packaged as incremental or additional functionality [Cameron *et al.* 1993; Areces *et al.* 2000] of usefulness [Hsi and Potts 2000] to system users and added to the basic system [Bond *et al.* 2004]; and encapsulates both functional and quality requirements [Sochos *et al.* 2004]. From a software designer's point of view this notion of a feature is likely to be more useful. However it fails to distinguish a

feature from other functionality organising concepts such as modules, functions, procedures, and classes.

A feature from a Requirements Engineering Perspective: The three notions of a feature discussed above are not very useful for reasoning about conflicts between features. This is because they are not explicit about what actually makes a feature. In Maccari and Heie (2005) [Maccari and Heie 2005] a feature is regarded as a unit of behaviour that is specified by a set of logically related requirements whose implementation has a tangible value to the user. Logically-related means requirements that are very much dependent on each other in such a way that it makes sense to have them implemented together.

Zhang *et al.* (2005) [Zhang *et al.* 2005b] makes a distinction between the *intension* and *extension* definition of a feature. Intension describes the intrinsic properties of a feature, in which a feature is viewed as a cohesive set of logically-related requirements. The extension view of a feature characterises the external embodiment of a feature by characterising it from a user's perspective. A concurrent view of these two concepts is in agreement with Maccari and Heie (2005) [Maccari and Heie 2005] that a feature is a set of logically-related requirements whose implementation is intended to deliver some tangible end-user value.

From the preceding discussion, a feature can be defined as a self-contained subset of system behaviour, designed as a cohesive chunk of functionality, which is user-accessible, and packaged as incremental or additional functionality intended to deliver a particular behavioural effect [Turner *et al.* 1999; Zave 2001; Calder *et al.* 2003]. This definition is overloaded and fuzzy. In this thesis we define a feature as a set of related *requirements* and their *specifications* intended to deliver a particular behavioural effect in a given *context*. Our definition of the concept of a feature is grounded on the entailment relation [Zave and Jackson 1997]. The entailment relation relates three sets of descriptions: requirement, specification, and context. It states that a specification satisfies a requirement given some assumption about

the behaviour of the context (W). While we do not claim the definition of a feature we have given above to be complete, we believe that it is more useful (than others given in the literature) when considering the feature interaction problem in a Requirements Engineering context. We elaborate on this in section 2.2. Our definition is also consistent with the clarification of the notion of feature given in Classen *et al.* (2008) [Classen *et al.* 2008].

Our view is similar to the concept of a *requirements module* proposed in Bredereke (2004) [Bredereke 2004] except that their definition does not explicitly consider the specification and context. They present a comprehensive comparison between the concept of a feature and that of a requirements module. A requirements module is described as a set of properties that are likely to change together. Meanwhile a feature is said to consist of properties selected to satisfy needs of a given stakeholder. Such properties need not have any similarities in terms of when they change.

Our definition makes explicit the specification and context of a feature - which we consider essential components in reasoning about conflicts. In being implicit about specification and context, the concept of a requirements module falls short when it comes to reasoning about conflict between requirements. Conflicts observed between requirements can be explained in terms of properties of the context and actions issued according to the specification. For example consider two requirements: (1) to regulate the temperature in a room by opening and closing the door; (2) to secure the room by ensuring that the door is closed and locked at night. These two requirements cannot be said to conflict until it is certain they are both referring to the same door. Even if we establish that they are referring to the same door, the conflict may not occur since it is dynamic – only certain to occur if both requirements need to be satisfied at the same time. This example illustrates the role of context in reasoning about conflicts between requirements.

The notion of a feature that we have adopted above raises another question: Given a set of requirements, how do we structure them into features? We discuss this in the next section.

2.1.2 Deriving features from Requirements

Given a set of requirements, what relationship should they have among themselves to be considered to belong to a particular feature? In this section we review requirements clustering – a common technique for deriving features from requirements.

Requirements Clustering: The idea of structuring requirements into clusters was first proposed by Hsia and Yaung (1988) [Hsia and Yaung 1988] and further developed by Hsia *et al.* [Hsia and Gupta 1992; Hsia *et al.* 1996] . Their main aim was to decompose a software system into manageable components delivered as system increments, thereby reducing the complexity of system development. They stated that clustering enables the sub-division of a large system's functionality "*into user-recognisable components where each component can be used, almost independently, to satisfy part of the user's needs*" [Hsia and Yaung 1988; Hsia *et al.* 1996]. Turner *et al.* (1999) [Turner *et al.* 1999] present a more detailed review of techniques on requirements clustering.

Requirements clusters may be regarded as features since features are also meant to support the paradigm of incremental functionality delivery [Keck and Kuehn 1998]. In Hsia and Gupta (1992) [Hsia and Gupta 1992] requirements clusters are formed based on similarities in the requirements generalised by Abstract Data Types (ADT). Turner *et al.* (1999) [Turner *et al.* 1999] argued that clustering requirements around ADT is a *solution space* concern and should not be encouraged as it forces design decisions very early in the *problem space*. They instead suggested that clustering should be done based on logical relations between the requirements which contribute to the properties of the feature to be realised. However, in Svahnberg *et al.* (2005) [Svahnberg *et al.* 2005] it is argued that identification of dependencies and

relationships between requirements and their subsequent grouping into features can in itself be considered as the first step towards a solution while in the problem space.

Awareness of the interdependencies between requirements is an important factor in the detection of feature interactions. This is because feature interactions often arise because of dependencies between requirements that a developer has not known in advance. Dahlstedt and Persson (2003) [Dahlstedt and Persson 2003] proposed a reference model of requirements interdependencies. They identify two broad interdependencies: *structural* and *cost/value*. *Structural interdependencies* group requirements according to hierarchical relationships and horizontal relationships. These include *requires*, *explains*, *similar to*, *conflicts with*, and *influences*. *Cost/value interdependencies* are concerned with the cost or value of implementing a requirement in relation to the impact of the implementation of that requirement on the cost or value for the user of another requirement. These include *increases/decreases cost_of* and *increases/decreases value_of*.

In the *increases/decreases cost_of* relationship, choosing to implement one requirement over another may either increase or decrease the cost of implementing the remaining requirement(s). For example, the cost of implementing subsequent requirements may decrease if a lot of functionality can be reused from previous implementations of other requirements. Hence the cost of implementation dependency is concerned with identifying requirements that should be implemented as a group since that would decrease the cost of their implementations. Similarly, in the *increases/decreases value_of* dependency, the implementation of one requirement may either decrease or increase its value to the user of another requirement. For example a requirement whose implementation improves the accuracy and speed of web search may have increased performance value to a user of a requirement that makes use of the web search results. Using the dependency relationships between requirements in order to determine the optimal order in which they should be implemented is the subject of *software release planning* [Ruhe and Saliu 2005].

While the model proposed by Dahlsted and Persson (2003) [Dahlstedt and Persson 2003] is useful as a starting point, it is not practical due to its higher level of abstraction. In particular it does not give practical guidance on: (1) how to identify and (2) how to describe/model requirements dependencies. Chen *et al.* (2005) [Chen *et al.* 2005] proposed a classification of requirements relationships based on *resource* and *functional* dependencies. Requirements have a *resource relation* if they share access to modification of the same property. Two requirements are said to have a *functional relation* if the satisfaction of one requirement depends on the correct behaviour of the other. Table 2.1 presents a summary of these requirements relationship classifications.

Table 2.1: Requirements Relationships Classification

BASIS FOR CLASSIFICATION	REQUIREMENTS RELATIONSHIP	CHARACTERISTICS	EXAMPLE	SOURCE
Resource or Object Relation	Strong Resource Relationship	Occurs between requirements that modify the same resource or share the same object.	Both Edit and Format features in a word processor modify the same document	[Chen et al. 2005] and [Hsia and Yaung 1988; Hsia and Gupta 1992]
	Weak Resource Relationship	Occurs between requirements that access the same resource but only one of them modifies it.	Cut and Paste features of a text editor. Cut writes text to the clipboard. Paste reads the text from the clipboard.	[Chen et al. 2005]
Functional Dependency	Necessity, Availability Require and Restrictive Relationships	The satisfaction of one requirement depends on the correct behaviour of the other, i.e., there is a functional connection between the requirements.	The functionality of Parameter Steering In modern cars depends on Speed Measurement. Parameter Steering is a speed-dependent steering system that reduces the steering effort required at lower speeds (e.g. during parking) and increases it at higher speeds.	[Chen et al. 2005], [Hsia and Yaung 1988; Hsia and Gupta 1992], [Yoo et al. 2004], and [Ferber *et al.* 2002]

In summary, creating requirements clusters depends on a number of factors which include functional, structural, cost, and value dependencies. Interesting questions arising from these classifications are: (a) What criteria can aid a requirements analyst in deciding which of these dependencies to apply in creating requirements clusters? (b) If more than one dependency is chosen to create a cluster, how do we balance between these dependencies?

2.1.3 Requirements Cluster Consistency

With respect to the feature interaction problem, the notion of a feature as a set of logically-related requirements raises the issue of consistency between clusters. More specifically, how should the structuring of requirements into features be done in such a way that feature interactions are minimised? According to [Gibson 1997; Gibson *et al.* 1999], one of the contributing factors to feature interactions is poor clustering of requirements into features. There is a lack of documented guidelines for creating requirements clusters (features) in such a way that the effects of the feature interaction problem are minimised. The proposed requirements clustering techniques by [Hsia and Yaung 1988], [Chen *et al.* 2005], and [Yoo *et al.* 2004] do not address this problem.

We noted in the discussion above that current definitions of a feature do not distinguish between the *requirement*, *specification*, and *context* – and are thus insufficient in reasoning about conflicts. In the next section we present the problem frames notation which we use to structure a feature in terms of the definition we proposed in section 2.1. We chose the problem frames notation due to its ability to explicitly distinguish between requirement, specification, and context.

2.2 Problem Frames

In section 2.1 we characterised a feature as having a requirement, specification and context. In chapter 1 we argued that context is important in reasoning about conflicts between features

because feature interactions manifest themselves on the shared context. In this section we introduce the problem frames notation which we will use to identify and analyse the relationship between the three sets of descriptions in a feature. Problem frames is based on the entailment relation and this makes it suitable for modelling features considering the definition of a feature stated in section 2.1.

2.2.1 The Philosophy of Problem Frames

Problem frames are an intellectual tool for analysing and structuring software development problems. The philosophy of problem frames is that some software development problems are recurring. Based on this premise, the idea is to document structures of commonly recurring problems and their solutions in problem-solution patterns. When a problem that matches a well known problem structure is encountered, the solution part of the pattern can then be re-used to solve the problem at hand.

Using the problem frames approach, we model a feature as a relation between three sets of descriptions: requirement (R), specifications (S), and *problem domains* (W) [Laney *et al.* 2007; Classen *et al.* 2008]. Problem domains represent properties that are part of the problem world such as resources. For example in a problem to design a controller to regulate the freshness of air in a room, a window would be a problem domain as opening it would allow fresh air into the room. We use the term *context* to mean a set of problem domains in a single feature. The behaviour of a problem domain is called its *domain description* [Jackson and Zave 1993]. Domain descriptions are indicative in that they express static relationships between the occurrence of *events* and the resulting effects in terms of *state* changes in the problem domain. In essence a domain description is a model of the dynamic behaviour of a real-world problem domain. For example *pushing a window towards its frame would lead to it being eventually closed.*

A requirement is stated in optative mode and describes desired states and behaviour of the problem world [Zave and Jackson 1997]. For example *the window should be opened between 6 am and 6 pm*. A specification satisfies a requirement by issuing actions which effect appropriate state changes on the problem domain. A specification is executed by a *machine* [Jackson 2001] such as the computer hardware on which an application is installed. In the problem frames approach, the argument that a specification satisfies a requirement is expressed through Jackson and Zave's *entailment relation* [Zave and Jackson 1997]. The entailment relation is: a specification satisfies a requirement given assumptions about the behaviour of the problem context. Formally, this relation is expressed as:

$$S, W \vdash R \qquad (2a)$$

WHERE "$\vdash$" is the entailment operator.

Using 2a we would express the security problem description as:

$$S_{sec}, W_{sec} \vdash R_{sec} \qquad (2b)$$

The entailment relation as expressed in 2a and 2b does not prescribe languages for expressing the three artefacts. This absence of prescription has the advantage of giving the requirements analyst freedom to choose a language of their choice for representing details of the three descriptions. In order to support the argument that 2b holds we need to know the details about the specific behaviour of S_{sec} and W_{sec}. One way to describe the three artefacts (S, W, and R) is in terms of *events* and *state* changes.

2.2.2 Modelling Features as Problem Descriptions

The relationship between the descriptions of the requirement, specification, and context is called a *problem description* [Jackson 2001]. A problem description can be represented graphically in a *problem diagram*. Figures 2.1 is a problem diagram documenting a problem description of securing a laptop on a desk in an office [Haley *et al.* 2008]. The security requirement, shown in the dotted oval, states that the office door and windows should be locked at all times except when unlocked by the office owner. The security machine, shown in the rectangle with double bars, describes the events that should be issued to the door and windows domains to satisfy the security requirement. Note that, for simplicity, we only project the *Door* and a *Security Card* problem domains in the diagram.

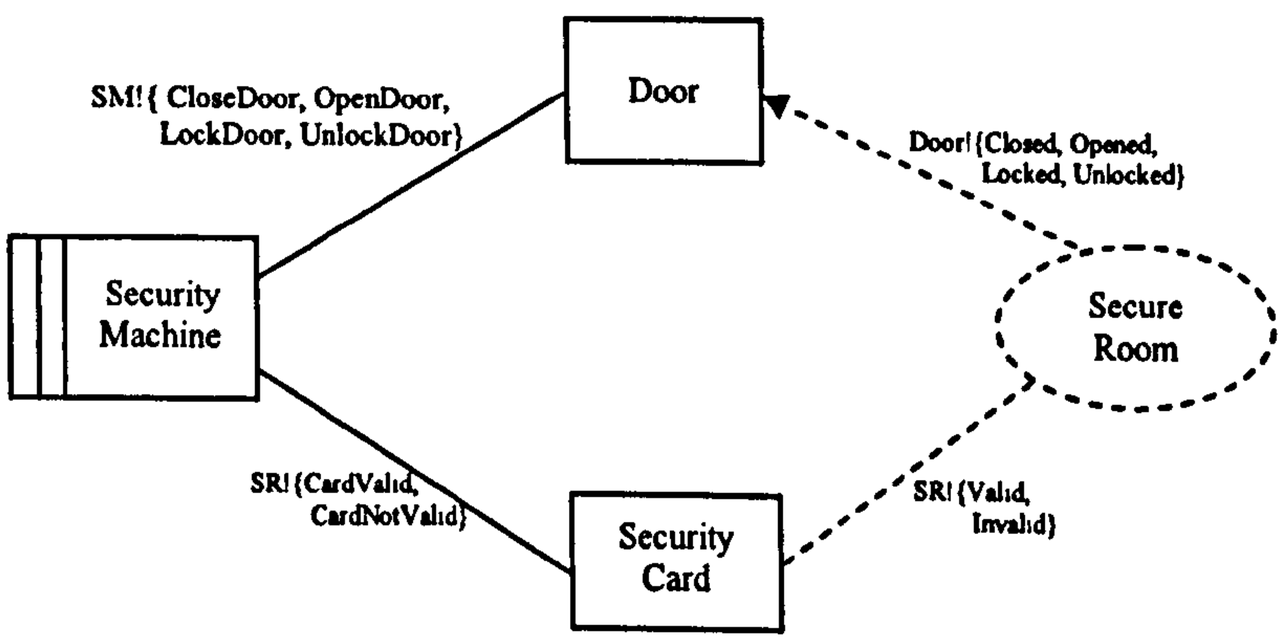


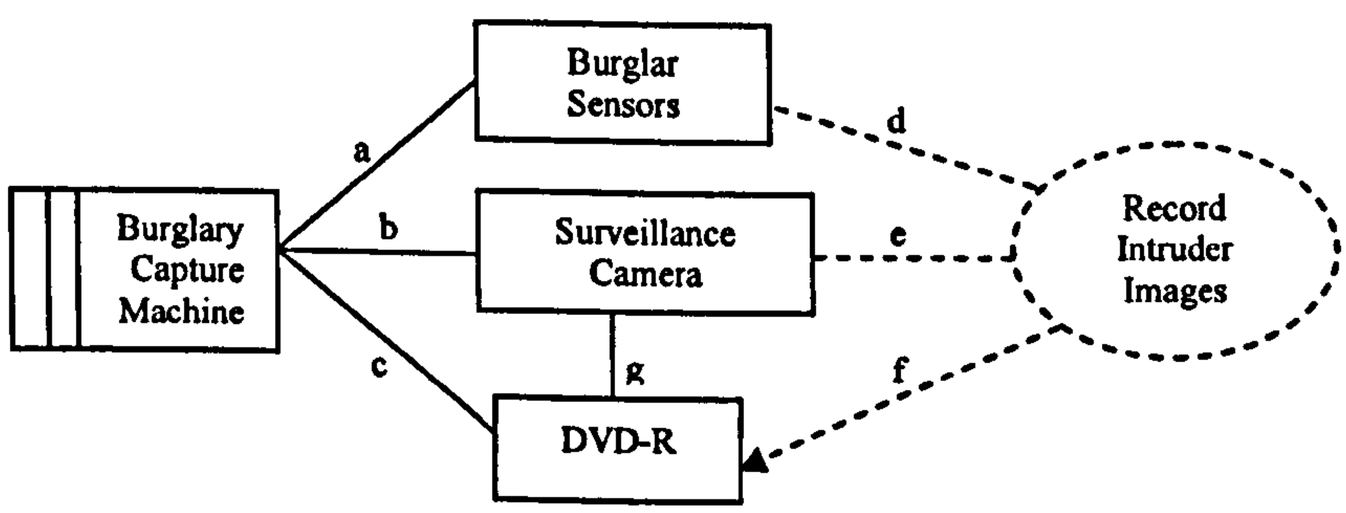
Figure 2.1 Office Security Problem Diagram

The Security Card problem domain represents an identity card which is validated by the Security Machine before opening the door. Labels on the interfaces between domains show the phenomena shared. The label also indicates which of the two domains (at either side of the interface), control the phenomena. For example, the label *SM!{OpenDoor, CloseDoor, LockDoor, UnlockDoor}* means that the events OpenDoor, CloseDoor, LockDoor, and UnlockDoor are controlled by the Security Machine. On the requirements interface the corresponding phenomena observed is whether the door is *Closed, Opened, Locked,* or *Unlocked.*

2.3 Smart Home Problem Descriptions

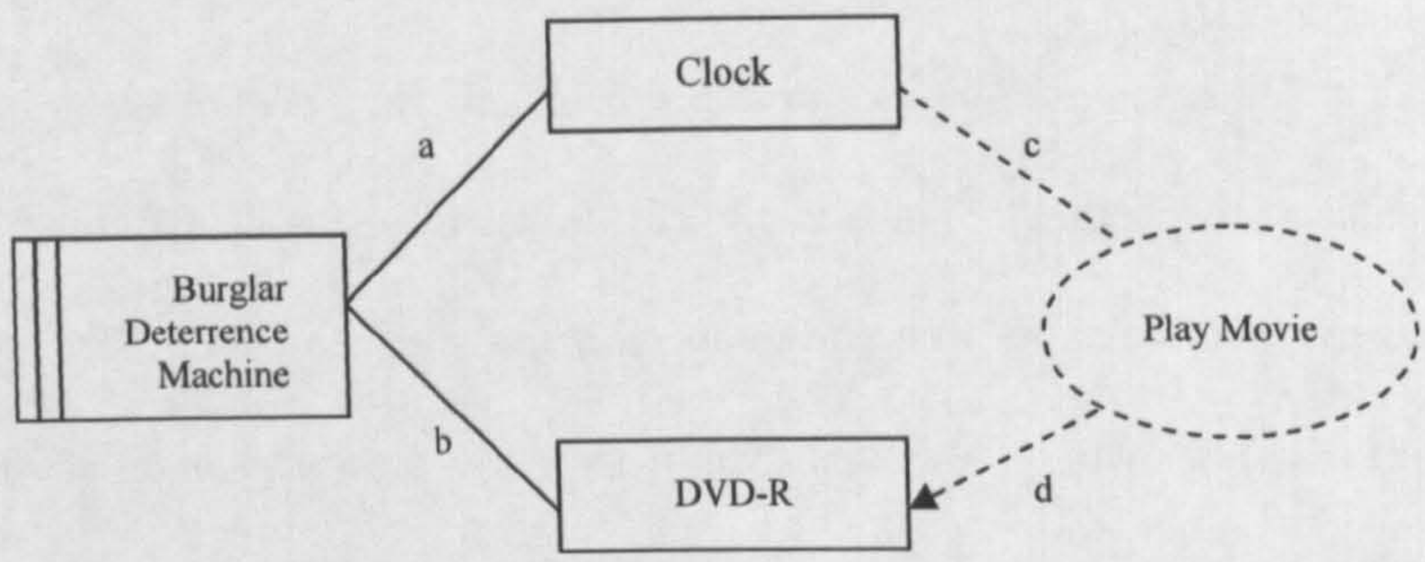
Figures 2.2 and 2.3 are examples of problem diagrams for *burglar deterrence* and *burglary capture* features, respectively. This example is drawn from the smart home domain. We use these features in the rest of the thesis to illustrate our approach. The burglary capture requirement (R_{cap}), shown in the dotted oval, is to "*record intruder images*". This requirement is satisfied by the *burglary capture machine (BCM),* shown in the rectangle with double bars, executing the capture specification (S_{cap}). The ability of the BCM to satisfy the capture requirement depends on assumptions about the behaviour of the three problem domains: burglar sensors, surveillance camera, and DVD-R. On detecting a burglary through the sensors, BCM instructs the DVD-R to record images of the intruders captured by the surveillance camera.

The problem diagram in Figure 2.2 also shows the interfaces between the machine, problem domain, and requirement. Interface *a* shows the phenomena that are shared between the burglary capture machine and the burglar sensors. The phenomena in this interface are *Movement_Signals*. The domain name before the "!" symbol indicates the domain that controls the phenomena. In this example *Movement_Signals* is controlled by the burglar sensors domain.




```
a: BS!{Movement_Signals}
b: BCM!{startCapture, stopCapture}
c: BCM!{start, stop, pause, play}
    DVD-R!{isRecording, isStopped, isPaused, isPlaying}
d: BS!{Movements}
e: SC!{isCapturing, isStopped}
f: DVD-R!{Recording, Stopped, Paused,  Playing}
g: SC!{Video_Signals}
```

Figure 2.2 Problem Diagram of Burglary Capture Feature




```
a: EM!{current_time_request}
   CL!{current_time_report}
b: BDM!{start, stop, pause, play}
   DVD-R!{isRecording, isStoped, isPaused, isPlaying}
c: CL!{current_time}
d: DVD-R!{Recording, Stopped, Paused, Playing}
```

Figure 2.3 Problem Diagram for Burglar Deterrence Feature

The burglar deterrence requirement (R_{det}) is to playback a movie from DVD media when the house owner is away to give the impression that someone is home - thus deter potential thieves from breaking-in. Composition of the two features is shown by the problem diagram in Figure 2.4. The two features share control of the DVD-R. This composition is expected to satisfy both the burglary capture and deterrence requirements. However, the DVD-R cannot record and playback at the same time, as shown by the state transition diagram in Figure 1.1. Hence, the deterrence and burglary capture requirements may not both be satisfied simultaneously. One scenario where these features can conflict is when a thief breaks-in at a time when a movie is playing.

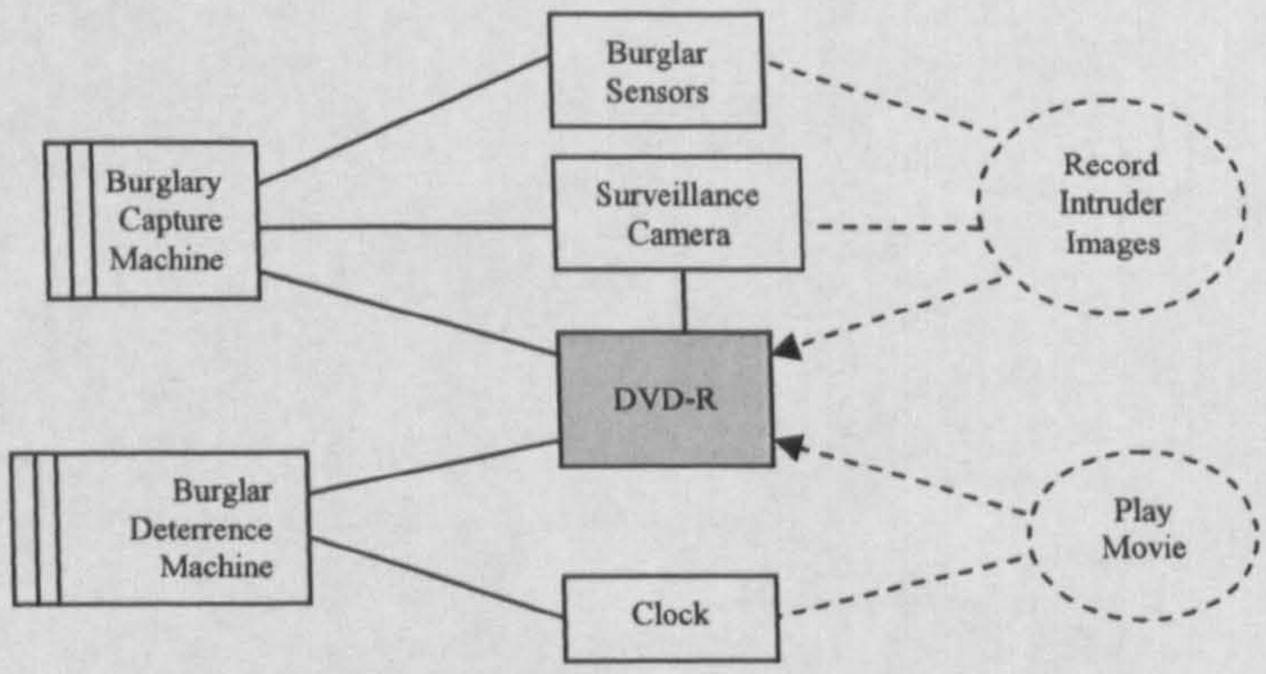


Figure 2.4 Composition of Burglary Capture and Deterrence Features

The inconsistency between these two features occurs because they are both trying to use the same DVD-R. Therefore, these features are inconsistent with respect to the DVD-R domain because they are trying to engage it in behaviours that are incompatible. Incompatible here means that due to the nature of the shared domain the two behaviours cannot happen simultaneously. This problem would not occur if each feature had its own DVD-R, or if the shared DVD-R used some time-division technique which enabled it to record and playback at the same time. This example illustrates that feature interaction is a *context sharing problem* [Nhlabatsi *et al.* 2008].

2.4 The Composition Controller Approach

The problem of sharing the DVD-R can be solved by composing the contingent specifications through an arbitrator. The arbitrator intercedes between the specifications of the features and the shared domain. In our approach we used a Composition Controller as an arbitrator. Using prioritisation, a Composition Controller filters events issued by the capture and deterrence machines. This prevents conflicts in the sharing of the DVD-R. Figure 2.5 shows a composition diagram of the burglary capture and burglar deterrence features with a Composition Controller.

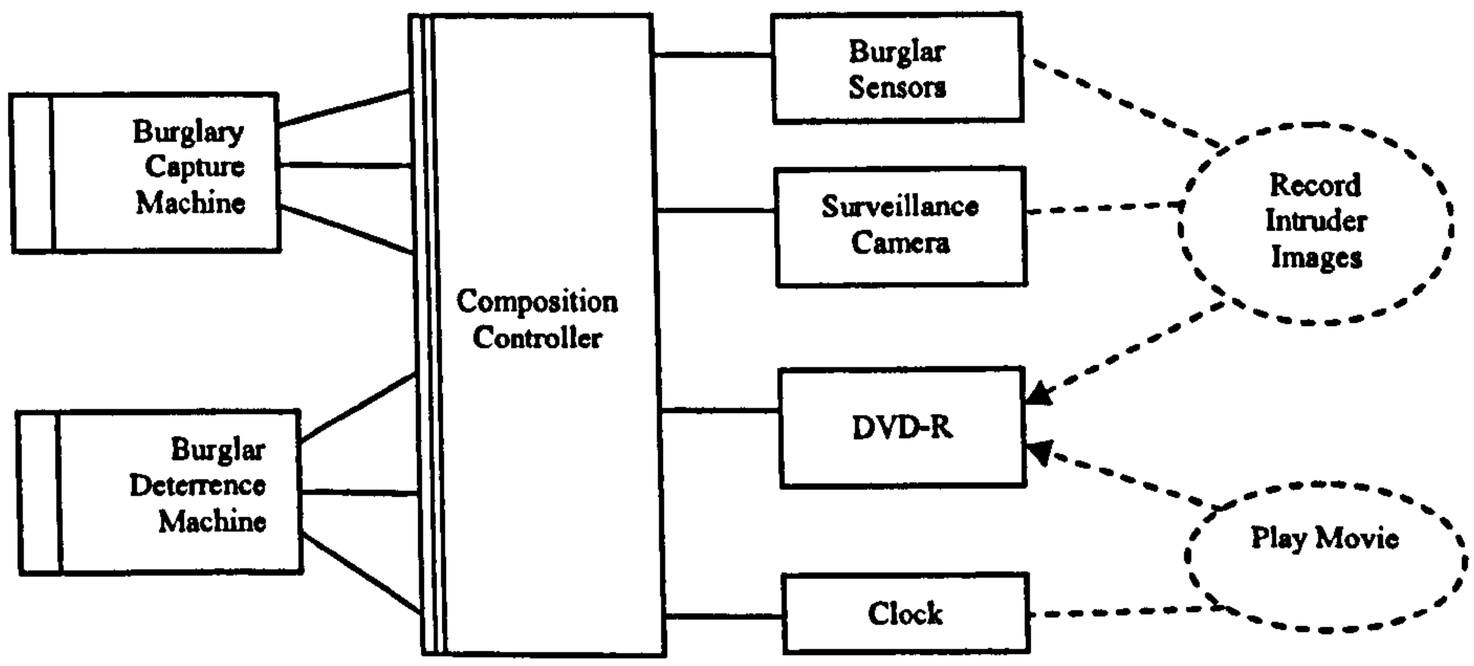


Figure 2.5 Composition of Capture and Deterrence Features with a Composition Controller

The Composition Controller approach makes the following assumptions:

i. Specifications are expressed in terms of events that cause state changes in the problem domain.

ii. The specification of each feature includes not only events that should occur for the requirement to be satisfied. It also includes events that should be rejected for the satisfaction of the requirement to persist over a given period.

The semantics of the Composition Controller are as follows: Requirements are prioritised and events issued by specifications with lower priority requirement are rejected in favour of higher priority requirements. With these properties a Composition Controller is able to resolve inconsistencies resulting from sharing a resource.

2.5 The Event Calculus

A problem diagram captures the scope of the problem to be solved by showing a static relationship between the requirement, specification and context. Such a static relationship is insufficient in facilitating a systematic derivation of a specification because it does not show the dynamic interactions between the *specification* and *context* that result in the satisfaction of the *requirement*. The entailment relation states that a specification assumes certain given properties about the behaviour of the context projected in a problem diagram. Therefore, to facilitate the derivation of a specification and to argue (formally) that the resulting specification satisfies the requirement, the dynamic behavioural properties of the context should be made explicit.

The Event Calculus (EC) is a logic system for reasoning about how the occurrence of events change the state of the world. We use the EC in this thesis to express domain descriptions and to facilitate the automation of deriving specifications. In this section we introduce the concepts of the EC that we use in the rest of the thesis.

2.5.1 Basic Constructs of the Event Calculus

The EC consists of three basic constructs: *events*, *fluents*, and *timepoints* [Mueller 2006a]. An event represents an action which may occur to a problem domain. For example, *pushing the door towards its frame* is represented as a *CloseDoor* event in the security problem shown in Figure 2.1. A fluent is a time-varying property describing the state of a problem domain such as *the door is closed*. A timepoint is an instant of time, for example 08:05:45 pm.

2.5.2 Event Calculus Predicates

A fluent is either true (holds) or false (does not hold) at a timepoint or over an interval. The occurrence of an event at a timepoint may change the truth value of a fluent. When an event results in a fluent being true it is said to *initiate* the fluent. Figure 2.6 is a domain description of the door in the security example given in section 2.2. According to WD_1 the occurrence of an *OpenDoor* event results in the *Opened* fluent being true. If the occurrence of an event means a fluent will be false then that event is said to *terminate* the fluent. For example, according to WD_5, a *CloseDoor* event terminates the *Opened* fluent. This results in the door being closed.

Initiates (OpenDoor, Opened, t)	[WD_1]
Initiates (ClosedDoor, Closed, t)	[WD_2]
Initiates (LockDoor, Locked, t)	[WD_3]
Initiates (UnLockDoor, UnLocked,t)	[WD_4]
Terminates (CloseDoor, Opened, t)	[WD_5]
Terminates (OpenDoor, Closed, t)	[WD_6]
Terminates (UnLockDoor, Locked, t)	[WD_7]
Terminates (LockDoor, UnLocked,t)	[WD_8]
Initially (Closed)	[WD_9]

Figure 2.6 Domain Description of a Door

A domain description models a real-world domain and forms a basis for reasoning about the behaviour of the modelled domain. It is therefore important that its behaviour is consistent with the actual state of the problem domain. In the EC all reasoning about future states is based on current states. The initial state of a problem domain is expressed with *Initially()* clauses. These state which fluents are assumed to be true when the problem domain is

initialised. For example, the domain description in Figure 2.6 assumes that the door is initially closed (WD_9). All other fluents not captured in the *initially* clause are assumed to be false (initially) and changes in their truth values are subject to the *commonsense law of inertia.* The commonsense law of inertia states that a fluent remains false until *initiated* and remains true until *terminated.* Table 2.2 shows the predicates of the EC we will use and their meanings.

Table 2.2 Event Calculus Predicates

Fluent	Description
Initiates(e,f,t)	Fluent f starts to hold after event e at time t.
Terminates(e,f,t)	Fluent f ceases to hold after event e at time t.
Initially(f)	Fluent f holds at time 0
Happens(e,t)	Event e occurs at time t.
HoldsAt(f,t)	Fluent f holds at time t.
Clipped(t1,f,t2)	Fluent f is terminated between times t1 and t2.

2.5.3 Event Calculus Meta-Rules

Based on initial conditions, events that have happened, and rules that state how fluents are changed when events happen (domain descriptions), it is possible to determine which fluents hold. This is summarised in the EC rules below.

$$\text{HoldsAt}(f,t1) \leftarrow \text{Initially}(f) \wedge \neg \text{Clipped}(0,f,t1) \qquad \text{[EC1]}$$

$$\text{HoldsAt}(f,t2) \leftarrow \text{Happens}(a,t1) \wedge \text{Initiates}(a,f,t1) \wedge (t1 < t2) \wedge \neg \text{Clipped}(t1,f,t2) \qquad \text{[EC2]}$$

$$\text{Clipped}(t1,f,t2) \leftarrow \exists a,t1\ [\ \text{happens}(a,t) \wedge \text{terminates}(a, f, t1) \wedge (t1<t<t2)] \qquad \text{[EC3]}$$

EC1 states that a fluent holds if it held initially and no event has occurred to stop it holding. EC2 states that a fluent holds if an event happened that makes it hold and no event has happened to stop it holding. EC3 states that if an event happens in the period between t1 and t2, and that event terminates fluent f, then f becomes false during that period. These three rules are referred to as meta-rules since they form the foundation of all reasoning about occurrence of events and resulting effects in the Event Calculus language.

2.6 Chapter Summary

This chapter has explored the concept of a feature from a Requirements Engineering perspective. We have defined a feature as a triplet; structured in terms of requirement, specification and context. Our definition of 'feature' is grounded on the entailment relation. The problem frames approach is also based on the entailment relation. For this reason we use the problem frames approach in modelling the structure of a feature.

In the entailment relation, it can be argued that a specification satisfies a requirement by showing that the dynamic behaviour of the specification changes the state of the context to that desired in the requirement. Hence deriving a specification involves taking into account the indicative properties of the context. The indicative properties describe the dynamic behaviour of context in terms of how the occurrence of events results in state changes. We have introduced the Event Calculus, a language for modelling the behaviour of the context and reasoning about the effects of events. In chapter 5 we show how to use this reasoning capability to derive contingent specifications.

Chapter 3. Related Work

The feature interaction problem has been studied in depth in the telecommunications domain.

This is evidenced in the conference proceedings [Calder and Magill 2000; Reiff-Marganiec and Ryan 2005] and special issue journals [Lograppo 1998; Akyildiz *et al.* 2000; Amyot and Logrippo 2004; Reiff-Marganiec and Ryan 2007] documenting research results on proposed approaches to addressing this problem. In recent work examples of feature interactions have been documented in other domains.

In general feature interaction is a conflict between features. As a result, approaches to addressing the feature interaction problem are similar to those proposed for analysing conflicts between goals [van Lamsweerde *et al.* 1998; van Lamsweerde and Willemet 1998], policies [Lupu and Sloman 1999; Dunlop *et al.* 2003; Reiff-Marganiec and Turner 2004; Blair and Turner 2005; Turner and Blair 2007], viewpoints [Easterbrook 1993; Easterbrook and Nuseibeh 1996], aspects [Rashid *et al.* 2002; Baniassad *et al.* 2006], and requirements [Robinson and Pawlowski 1999] in inconsistency management.

Approaches addressing this problem can largely be divided into three categories: design-time, run-time, and hybrid [Keck and Kuehn 1998; Hall 2000a; Calder *et al.* 2003; Hall 2005; Wilson *et al.* 2005; Nhlabatsi *et al.* 2008]. A common characteristic of feature interactions is that they result from sharing of context and are often subtle in nature. As an illustration consider the following example from the automobile domain:

> Consider a car which has an alarm system (*security feature*) and a crash protection system with air bags (*safety feature*). The alarm system enforces security of the car occupants and their valuables. When activated it ensures

that the doors and windows are locked; and monitors the state of the doors; and reports any burglary activity by activating the siren. Meanwhile, the safety feature ensures that in case of a crash, there is minimal loss of life. It achieves this by unlocking all doors in the event that an impact occurs on the front bumper.

> Let us consider a scenario where these features could interact. Assume the car is stationery at a traffic intersection with all doors locked by the *Security* feature. If a thief hits the front bumper with a big hammer, the *Safety* feature will unlock the doors allowing the thief to gain entry into the car.

This conflict demonstrates the subtle nature of the feature interaction problem. This feature interaction may not be obvious to detect until a scenario such as the one above occurs. In this example we have assumed that safety has a higher priority than security. Without such priority a non-deterministic behaviour would result as both features try to gain control of the doors.

The example highlights two problems. The first problem is how to detect, during composition, that satisfying the safety requirement will compromise the security requirement and vice versa? How do we detect during composition that having both security and safety features share control of the doors would lead to an undesirable interaction?

In the event of a genuine crash it is desirable for safety to compromise security as this may enable emergency personnel to get to the passengers in time. However, this is undesirable in the case of a crash 'simulated' by a burglar. Thus, the second problem is, once we know that safety compromises security, how to redesign the safety feature so that it is possible to differentiate between a real and faked accident. The redesign may involve taking into account

contextual properties that were not considered initially. This thesis does not deal with this kind of problems.

Approaches addressing the feature interaction problem can largely be divided into two categories: offline [Calder and Miller 2001; Felty and Namjoshi 2003; Calder and Miller 2006] and run-time [Velthuijsen 1993; Tsang and Magill 1998; Pang and Blair 2002; Kolberg and Magill 2007; Laney *et al.* 2007]. In this chapter we review these approaches from a Requirements Engineering perspective, with a focus on context sharing as a source of conflict between features.

In section 3.1 we advance the argument (introduced in chapter 1) that feature interaction is a context sharing problem by providing supporting evidence from the literature in the form of taxonomies and sources of feature interactions. Section 3.2 reviews approaches to managing conflicts offline while section 3.3 reviews runtime approaches. In section 3.4 we conclude the chapter by summarising the limitations of current approaches in addressing the initialisation problem.

3.1 Feature Interaction as a context sharing problem

In this section we show that context is at the core of reasoning about the feature interaction problem. We support this argument using the entailment relation, feature interaction sources, and feature interaction taxonomies. In chapter 2 we characterised a feature as a set of related *requirements* and their *specifications* intended to deliver a particular behavioural effect in a given *context*. Using this characterisation we show, in section 3.1.1, that when features are composed they enter into a relationship which is established through the context they share.

We review sources of feature interactions that have been documented in the literature in section 3.1.2. Our review concludes that: a conclusion that two requirements are in conflict is reached through a consideration of the behaviour of the context they share. In section 3.1.3 we

conclude our argument on the role of context by reviewing proposed taxonomies of feature interactions with respect to context. Finally, we present a summary of the discussion on feature interactions as a context sharing problem in section 3.1.4.

3.1.1 Formalisation of Feature Interaction through the Entailment Relation

When a feature interaction occurs, at least one of the requirements satisfied in isolation by the features in the composition may be violated. Figure 3.1 expresses the relationship two interacting features using the entailment relation [Jackson 2001] and the parallel composition notation[Abadi and Lamport 1993]. This is similar to the formal framework for feature interaction proposed in Godskesen (1995) [Godskesen 1995].

$S_1, W_1 \vdash R_1$ (1)

$S_2, W_2 \vdash R_2$ (2)

$S_1 || S_2, W_s \vdash R_1 \wedge R_2$ (3)

WHERE

$W_s = \{W_1, W_2\}$, $W_1 \in W_s$, and $W_2 \in W_s$

"{..}" is the set operator

"||" is the parallel composition operator

"∧" represents the logical AND operator

Figure 3.1 Feature interaction expressed using the entailment relation

The basic idea is that if a feature specification S_1 satisfies a requirement R_1, assuming context W_1 (1), and a feature specification S_2 satisfies a requirement R_2, assuming context W_2 (2); then it is desirable that their parallel composition satisfy the conjunction of R_1 and R_2 (3). Feature interaction occurs when there are shared properties between W_1 and W_2 whose relationship is such that (3) is not true. In general we call such properties the shared domain (W_s). When considered together the relations expressed in (1), (2), and (3) state that if each feature behaves correctly and satisfies certain properties in isolation, then it is desirable that it behaves correctly and continue to satisfy its requirements in composition with other features [Abadi and Lamport 1993].

In illustrating the concept of a shared domain we use an example adapted from [Laney *et al.* 2007]. Consider the office security example presented in Chapter 2. Recall that the security requirement is to control access to the office by opening the door only when a legitimate security card is presented. Assume that we now have a new requirement for a *climate control* feature. The climate control feature maintains the office at a cool temperature by opening the door when the temperature outside the office is cooler than that inside and closing it otherwise. Figures 3.2 and 3.3 show problem diagrams for the security and climate control features, respectively. Composition of the two features is shown in Figure 3.4.

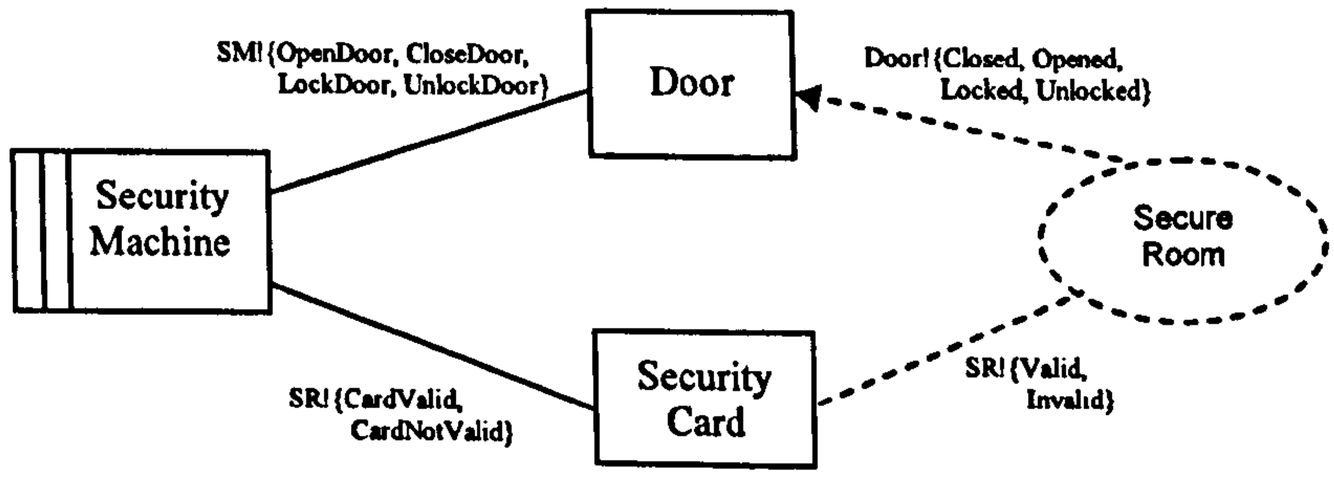


Figure 3.2 Problem Diagram of Office Security Feature

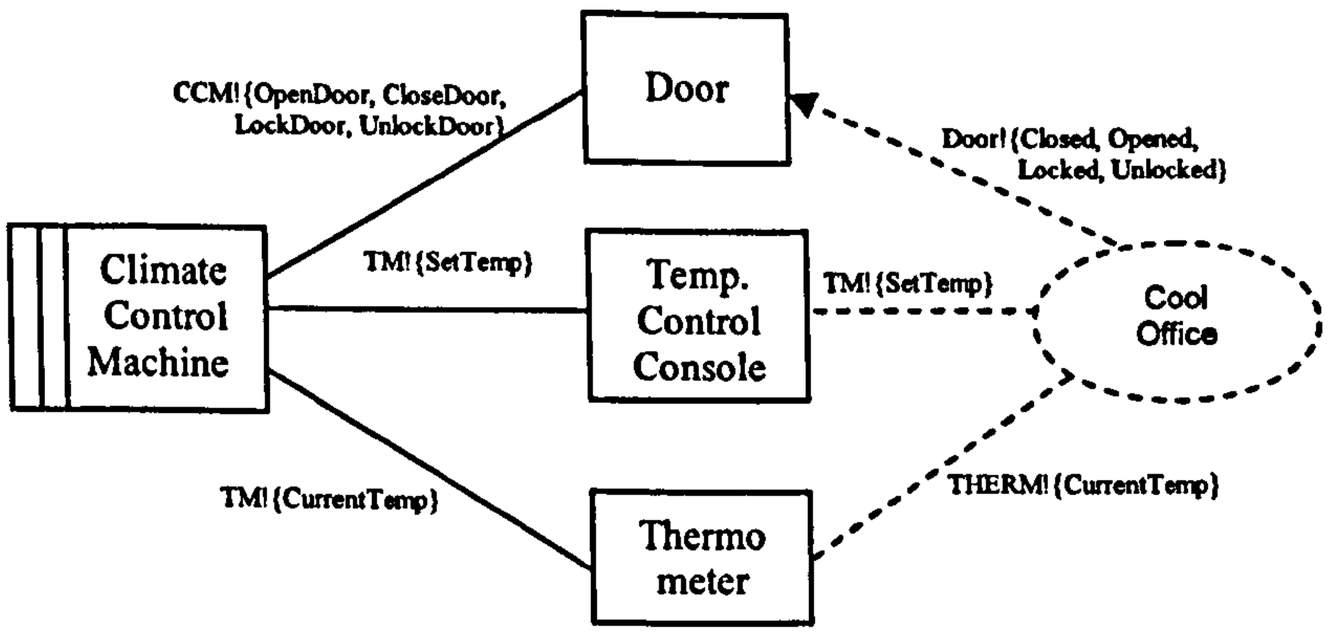


Figure 3.3 Problem Diagram of Office Climate Control Feature

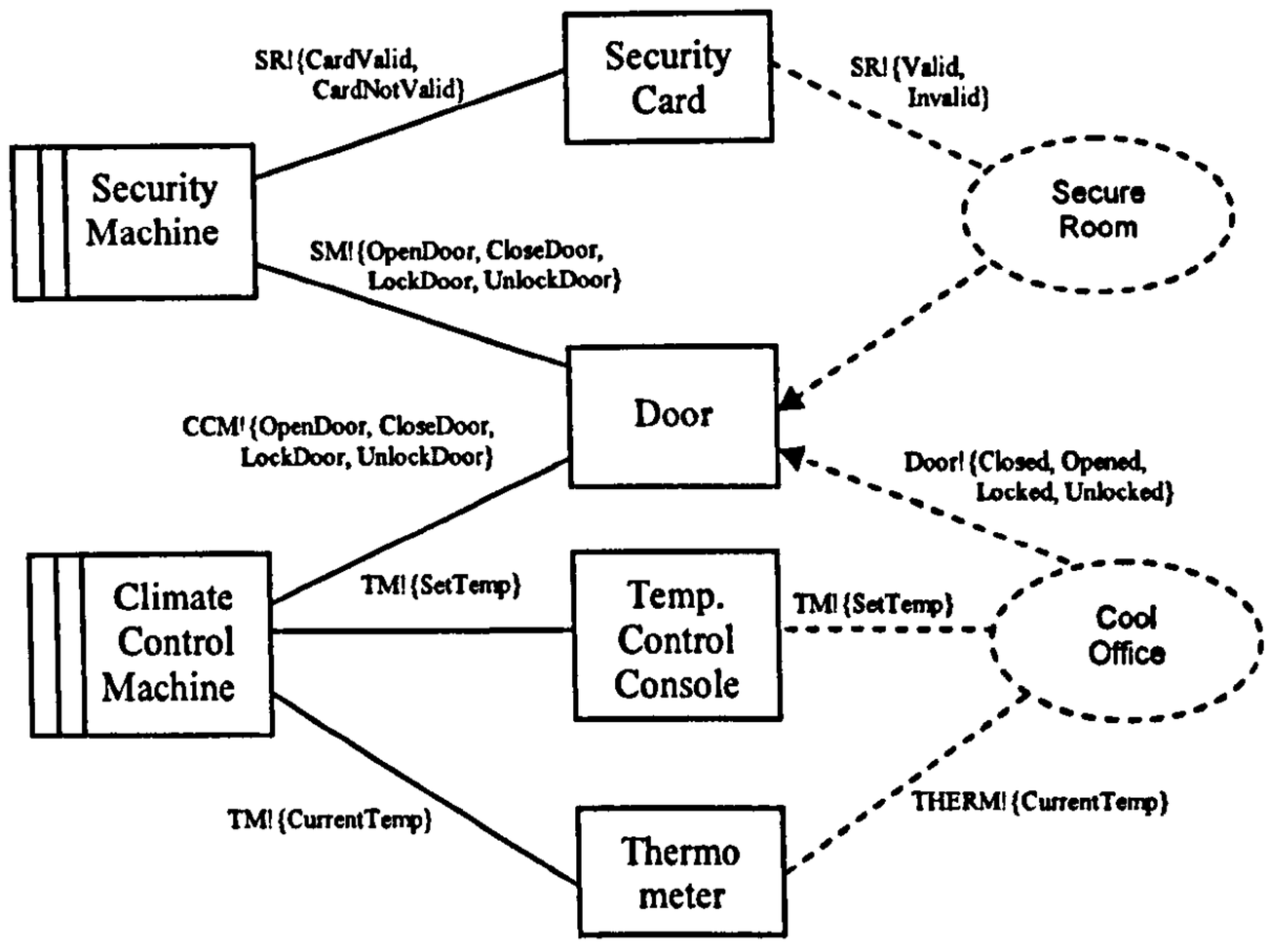


Figure 3.4 Composite Problem Diagram of Office Security and Climate Control Features

The two features share control of the door. Their composition is expected to satisfy both the security and climate control requirements. However, satisfaction of both requirements may not always be possible. For example satisfying the security requirement will be violated when climate control opens the door if a thief enters the room. According to the security requirement the door should only be opened upon presentation of a valid and legitimate security card. However, climate control opens the door without such security authorisation. This is a loophole in the security of the room. The conflict illustrated in this example arises because the two features share control of the door. We discussed the implications of the feature interaction problem for security requirements in Nhlabatsi *et al.* [Nhlabatsi *et al.* 2008].

3.1.2 Sources of Feature Interactions in Requirements

Feature interactions can be characterised by their causes. Table 3.1 shows a summary of some causes of feature interactions that can be attributed to relationships between requirements. This is based on a taxonomy of feature interaction sources proposed by Cameron *et al.*

[Cameron *et al.* 1993]. These include overlapping pre-conditions, requirements inconsistency, conflicting goals, violation of assumptions, and resource contention.

The way the sources of interactions are discussed in Cameron *et al.* [Cameron *et al.* 1993] gives the impression that they are independent. We argue that they should not be viewed in isolation. They are dependent on each other and they are parts of one whole. Only when they are considered together can the characteristics of feature interaction be fully appreciated.

Table 3.1: A Summary of sources of feature interactions (from [Cameron *et al.* 1993])

Source	Examples
Overlapping Pre-Conditions (non-determinism)	Call Waiting (CW) with Call Forwarding on Busy (CFB) and Voice Mail when Busy (VMB): All three features are triggered by the same pre-conditions (busy subscriber) but they perform different and contradictory actions on behalf of the same user. Note: features without overlapping pre-conditions could also interact during their execution because of inconsistencies in their post-conditions.
Requirements Inconsistency	Security (anti-theft system) and Safety (door un-locking in case of crash): Similar to above. Both security and safety share the same sensors and are hence triggered by the same conditions, but the actions they perform as a result are inconsistent with each other.
Conflicting Goals	Calling Line Identity Presentation (CLIP) and Calling Line Identity Restriction (CLIR): CLIP delivers the calling subscribers identity, while CLIR does the opposite. This manifests itself as conflicting goals when used by two subscribers.
Violations of assumptions	Calling Number Delivery (CND) and Unlisted Number (UN): Similar to the interaction between CLIR and CLIP. CND delivers the number of the calling subscriber to the called subscriber for identification, while UN prevents an unlisted subscriber number from being delivered to a called subscriber – an example of violation of data availability assumptions.
Resource Contention	Burglary capture and burglar deterrence features in a smart home. The capture feature records Images of the intruder on the VCR when a burglar is detected. The deterrence feature records Channel 4 news from 7:00pm to 8:00pm on the VCR. If a burglar breaks-in at 7:30pm the capture feature will not be able to record the intruder images since the VCR is already being used by the deterrence feature.

For instance, *Overlapping Pre-Conditions* means that the features involved are triggered concurrently. However, that they are triggered concurrently is a necessary but not a sufficient condition for a conflict. We need to take into account whether they have *Conflicting Goals* or (and) *Inconsistent Requirements*. To reach the conclusion that two goals are conflicting or two

requirements are inconsistent the behaviour of the *context* should have been taken into account.

As an illustration, the conflict between Burglary Capture and Burglar Deterrence (introduced in chapter 1) is due to the nature of the DVD-R being incapable of recording and playback concurrently. The conflict on the DVD-R is a classic example of *resource contention* [Bisbal and Cheng 2004]. If the DVD-R had the capability to record and playback concurrently then overlapping preconditions of the two features would not lead to a conflict. Similarly, inconsistency between the two features would not be realised if they had no overlapping pre-conditions.

Composition may also *violate assumptions* made about the context when each feature is considered in isolation. For example in the office security and cooling problem discussed in section 3.1.1 the designer of each feature may have thought that the feature will have sole control of the door. This example illustrates how assumptions about the context made in isolation can be made invalid by composition - resulting in violation of feature requirements. The preceding discussion illustrates that context is at the core of the feature interaction problem.

In section 3.1.3 we discuss different taxonomies of feature interactions characterised (in part) in terms of the feature interaction sources discussed. The aim of feature interaction taxonomies is to establish relationship between sources and types of feature interactions [Kolberg *et al.* 2003; Hamed and Al-Shaer 2006; Shehata *et al.* 2007a; Shehata *et al.* 2007b]. Such relationships support detection of feature interactions through inspection of the correlation between their requirements and the nature of the context they share. For example if there is an overlap between the preconditions of two features then a non-deterministic feature interaction may occur (see second row in Table 3.1). Documentation and formalisation of these relationships provide information for early detection of feature interactions and aids

automation of the detection techniques [Braithwaite and Atlee 1994] [Lin and LIN 1994] [Sefidcon and Khendek 2000].

3.1.3 Feature Interaction Taxonomies

The current trend in feature interaction research is to study the problem in specific domains by applying generic feature interaction detection approaches. This has led to the creation of feature interaction taxonomies of the respective domains. Examples can be seen in reports on feature interaction in smart homes [Kolberg *et al.* 2003; Nakamura *et al.* 2004a], electronic mail systems [Hall 2000b], SIP services [Bond *et al.* 2004; Chi and Hao 2007; Kolberg and Magill 2007; Wu and Schulzrinne 2007], web services [Weiss and Esfandiari 2004; Weiss *et al.* 2007], embedded systems [Metzger and Webel 2003; Metzger 2004], policy-based systems [Dini *et al.* 2004; Reiff-Marganiec and Turner 2004; Blair and Turner 2005; Hamed and Al-Shaer 2006; Turner and Blair 2007], and product lines [Thiel *et al.* 2001; Bredereke 2005].

Although there are similarities between the feature interactions detected in these domains, each domain presents unique challenges. This has made it necessary to extend standard approaches to deal with specific types of feature interactions. For example, it has been shown that a consideration of the environment when addressing feature interactions in embedded systems is important [Kolberg *et al.* 2003; Metzger 2004]. This is because the environment creates dependencies between features which otherwise seem independent.

One of the earliest feature interaction type taxonomies was proposed by Cameron and Velthuijsen [Cameron and Velthuijsen 1993]. They identified four types of feature interactions that could occur in the telephony domain. Studies on feature interactions in other domains have resulted in the refinement of this taxonomy. A notable example is the taxonomy of feature interactions in smart homes proposed by Kolberg *et al.* [Kolberg *et al.* 2003].

Recently, these two taxonomies have been synthesised by Shehata *et al.* [Shehata *et al.* 2007b] to a generic taxonomy which the authors claim to be applicable to most domains. This taxonomy consists of 9 feature interaction types (shown in Table 2a).

This thesis uses a reduced version of the taxonomy proposed in Shehata *et al.* [Shehata *et al.* 2007b]. We merged S2, S3, S5, and S8 (see Table 3.2a) to form a *bypass* category. All these categories involve a conflict in which the execution of one feature fails due to an undesirable change of state (or failure to change state) of the shared context by another feature. S4 was merged with S6 to form a single *negative impact* category. Both categories refer to a case in which two features execute to completion but their post-conditions negatively interfere with each other.

Table 3.2a Shehata et al's Taxonomy

Interaction Types	Short Description
[S1] Non-Determinism	Occurs when two features have the same pre-conditions but different post-conditions.
[S2] Dependence	The execution of one feature depends on the correct execution of another.
[S3] Override (Same pre-conditions)	Occurs when two features that have the same pre-conditions have post-conditions for one feature that change the state of the context in such a way that the other feature does not finish executing.
[S4] Negative Impact (Same pre-conditions)	Occurs when two features with the same pre-conditions but with post-conditions that diminish each other.
[S5] Override (Linked trigger events)	Occurs between features that have linked trigger events but where the post-conditions of one feature change the state of the context in such a way that the other feature does not finish executing.
[S6] Negative impact (Linked trigger events)	Occurs between features that have linked trigger events with post-conditions that diminish each other.
[S7] Invocation Order	The behaviour of execution of the features in one sequential composition is different from behaviour when the sequential composition is changed.
[S8] Bypass	The execution of one feature prevents the execution of another by putting the context in a state that is different from the pre-conditions of latter feature.
[S9] Infinite Looping	Occurs when features execute indefinitely by continuously triggering each other.

The reduction leaves 5 categories, namely: Non-determinism, Negative Impact, Invocation Order, Bypass, and Infinite Looping (shown in Table 3.2b). We consider these categories to

represent conflicts that are at the core of the feature interaction problem. This thesis does not add any new categories to this taxonomy. It provides illustrations of the role of context in characterising these types of feature interaction.

Table 3.2b Reduced Taxonomy

Interaction Types	Short Description
Non-Determinism	Same as S1
Negative Impact	Merged S4 and S6
Invocation Order	Same as S7
Bypass	Merged S2, S3, S5 and S8
Infinite Looping	Same as S9

In doing so we use the concepts of *pre-condition*, *prestate*, *trigger event* and *post-condition* as used in [Shehata *et al.* 2007b]. *Pre-conditions* describe the conditions that should be true before a feature can execute. A pre-condition consists of sets of *prestates* and *trigger events*. A *prestate* describes what the state of the context should be before a feature can execute. When a *trigger event* occurs and the *prestates* are true, a feature is executed. *Post-conditions* describe the state of the system after the execution of the given feature. In essence, a post-condition describes the effect of executing a given feature. For this reason it is stated in optative mode [Zave and Jackson 1993]. For example Call Forwarding on Busy (CFB) in telephony is executed when there is an *incoming call* (trigger event) while the subscriber is *engaged on another call* (pre-state) and *forwards the incoming call* to a pre-specified number (post-condition).

Non-determinism: Non-determinism occurs when two or more feature specifications require a shared domain to engage in different behaviours simultaneously, when the domain can engage in only one of the requested behaviours at a time [Cameron and Velthuijsen 1993]. By domain we mean a property of the environment that a specification of a feature uses to satisfy the requirement(s) e.g. the Door in Figures 3.2 to 3.4. It becomes non-deterministic as to which of the required behaviours the domain should engage in. Non-determinism results from

overlapping pre-conditions with *inconsistent post-conditions*. As stated in section 3.1.2, overlapping pre-conditions makes it possible for features to be activated at the same time. Such overlaps may either be exactly or partially matching pre-states and trigger events.

The inconsistency of post-conditions can mean one or both of two things: a logical inconsistency between the individual feature requirements; and (or) inconsistency of the actions being performed on the problem domain. To illustrate this point, consider a DVD-R that is designed in such a way that it does not allow the functions of recording and playback to happen simultaneously.

Two features, F_1 and F_2, with requirements to record and playback, respectively, cannot be said to be inconsistent until we can ascertain that they are both trying to use the same DVD-R. Therefore, these features are inconsistent with respect to the shared context because they are trying to engage it in behaviours that are incompatible. This problem would not occur if: each feature had their DVD-R; or if the shared DVD-R used some time-division technique which enabled it to record from more than one video source at the same time. This highlights the feature interaction problem as a "context sharing problem" as illustrated in section 3.1.1

Negative Impact: Similar to a non-deterministic interaction, in this type of interaction, features have overlapping pre-conditions. The difference is that in this case both features are executed but the impact of their post-conditions are inconsistent [Cameron and Velthuijsen 1993]. The post-conditions of one feature diminish the effects of the post-conditions of the other feature. This type of interaction can manifest as a *resource contention* [Bisbal and Cheng 2004] or *inconsistent state changes* on a shared resource (such as device in a smart house [Kolberg *et al.* 2003]).

For example, consider two features: *AirFreshMonitoring* and *ClimateControl*. The requirements for the *AirFreshMonitoring* is that when the air quality in the room is poor and it

is day time the windows should be opened to refresh the room. The requirement of the *ClimateControl* feature is that during daytime the temperature in the room should be maintained at 25°C at all times by opening and closing the windows. Consider a scenario in when the air quality in the room is poor, the room temp is currently at 25°C, and it is too cold (or too hot outside the room). In response to the poor air quality, *AirFreshMonitoring* will open the window and this will either decrease or increase the room temperature. This has a negative impact on the requirement to maintain the room temperature at 25°C. Note that in this example satisfaction of the requirement is not immediate. The conflict arises when one of the features immediately close or open the windows while the requirement of the other feature is in the process of being satisfied.

Invocation Order: An invocation order interaction occurs when the sequential composition of two or more features result in different system behaviours under different sequential compositions [Shehata *et al.* 2007b; Weiss *et al.* 2007]. Two features F_1 and F_2 are said to be sequentially composed if at the end of the execution of F1, the execution of F2 is started. Sequential composition can be either implicit or explicit.

With *implicit sequential composition* the sequence of feature execution results from *linked trigger* events. Two events, e_1 and e_2, are linked if the occurrence of one event leads to the occurrence of the other. For example, consider two features associated with the control of an automated door, a *DoorOpenClose* feature and a *DoorLocking* feature. The *DoorOpenClose* feature controls the opening and closing of the door. When the door is opened and a *close* event occurs, the door starts *closing* and eventually generates a *closed* event when fully closed. If an *open* event occurs while the door is closed it starts opening and generates an open event when the door is fully opened.

The *DoorLocking* feature controls the locking and unlocking of the door. It locks the door 3 seconds after the occurrence of a *closed* event and locks the door immediately when a *lock*

event occurs. Similarly, this feature unlocks the door immediately when an *unlock* event occurs. *Open, Close, Lock* and *Unlock* events are generated by the user intending to enter the house where the door is mounted. This relationship between the events means that the execution of the *DoorOpenClosed* feature eventually leads to the execution of the *DoorLocking* feature. Hence the features have an *implicit sequential composition.* An invocation order interaction between the two features is illustrated below:

Consider a scenario in which the door is initially opened. Assume the door has close and open buttons which generate *close* and *open* events, respectively. When the user presses a close button the door is closed by the *DoorOpenClose* feature and eventually locked by the *DoorLock* feature. Define B1 to be this system behaviour.

Assume a second scenario in which the door is initially opened and the user issues a lock command which attempts to lock the door. This does not have an effect on locking the door since it is opened. If we assume that the type of lock used is mechanical then we can imagine the locking bar of the mortise lock protruding after the lock event is issued. If the user presses the close button the door will start closing but will not be able to fully close because of the protruding locking bar. Define B2 to be this system behaviour.

In the former scenario both features have executed properly and satisfied their requirements. However, in the latter scenario although both features have executed, none has satisfied their requirements. In B1 the door is properly closed and locked, but in B2 the door is left unclosed! Since B1 $\neq$ B2, then the composition of the safety and security features exhibits an execution order interaction.

Explicit sequential composition is the type of composition in which the preceding feature in a sequential composition is designed in such a way that it directly starts the execution of the

next feature. For example a security feature that automatically alerts the police through a phone call when a break-in is detected in a smart house is explicitly sequentially composed with a communications feature.

Bypass: One feature (F_1) *bypasses* another feature (F_2) if it changes the state of a shared context in such a way that F_2 is prevented from executing or completing execution (if already started). The new state of the context does not match that expected by F_2. As a result its requirements are never satisfied. To illustrate a *bypass* consider a scenario in which features F_1 and F_2 have linked trigger events. Assuming F_1 is triggered and executes to completion. Also assume that its post-conditions are different from the prestates of F_2. This means that when the trigger event of F_2 occurs, F_2 will not be executed since the current state of the shared context does not meet its prestates because of the execution of F_1.

For example consider a *Power Management* feature and a *Security* feature. The Power Management feature controls power consumption. It has parameters for monitoring the total power consumed and the rate of consumption. The total amount of power, measured in Kilowatts has a monthly limit. This feature has *adaptive power control* which ensures that power consumption does not exceed the monthly limit. Adaptive power control achieves this by monitoring and adapting power consumption by 'greedy' appliances. When an appliance consumes power at a rate higher than the average rate then that appliance is switched-off to ensure a steady consumption of power. On detecting a burglary, the Security feature raises an alarm by sounding a motorised siren. Assume a burglar is detected and the security feature starts the motorised siren which consumes power at a rate higher than the average rate. On detecting this, the power management feature switches off the power to the siren. As a result the security requirement is not satisfied and the power management feature is said to have bypassed the security feature.

Infinite Looping: Infinite looping feature interactions are unique in the sense that they defy the general notion of feature interaction. In this type of feature interaction individual feature requirements are not violated. A looping interaction occurs when two features are reciprocally linked in their post-conditions and trigger events [Cameron and Velthuijsen 1993; Kolberg *et al.* 2003; Shehata *et al.* 2007b]. Two features, F_1 and F_2, are reciprocally linked if the post-conditions of F_1 create the trigger events of F_2 and vice versa. To illustrate a looping interaction, assume that F1 is triggered and starts executing and creates the trigger events of F2. Feature F2 starts executing and in turn creates trigger events for F1. This process is repeated indefinitely - creating infinite looping.

For example consider a *Cooling* feature and a *Security* feature. When the temperature inside a house is higher than that outside, the Cooling feature opens the windows and starts the fan. On detecting movements in the house the security feature raises an alarm by sounding the siren and secures windows to ensure that the burglar does not get away. Consider a scenario in which the temperature in the house is hotter than outside. This triggers the *Cooling* feature which by starting the fan creates movements in the house which are interpreted by the *Security* feature as being caused by a burglar. The security feature shuts the windows. This makes the room warm again and triggers the Cooling feature, which again starts the fan and opens the windows. This cycle continues indefinitely.

The unconventional and subjective nature [Hall 2000a] of looping interactions has led to some researchers working on the feature interaction problem to argue that the general notion of feature interaction as presented in Figure 3.1 is not sufficient. Hall [Hall 2005] showed that in email systems the assertion that feature interactions only occur as a violation of individual feature requirements does not hold. He showed that feature interactions in this domain can occur without violation of individual feature requirements. For example:

Consider the interaction between *AutoResponder* and *GroupMail* features. The *AutoResponder* feature enables automatic response to incoming email messages when the addressee is away. The *GroupMail* feature enables the creation of a virtual group of email users in a domain. For example "PostGraduateStudents" could be a group of email addresses of all post graduate students in an institution. Let the email address of this group be *postgraduatestudents@open.ac.uk*. When a message is sent to the group address, it is forwarded to each of the affiliated addresses. Now let Armstrong be a member of the above group.

Consider a scenario where Armstrong is on vacation and an email is delivered to the group. The *GroupMail* feature sends this email to all addresses in the group. The *AutoResponder* feature replies by sending a response to the group email (not the originating address). When the *GroupMail* feature receives this message it forwards it to all the affiliated members. The cycle is repeated indefinitely. Note that both features have satisfied their individual requirements. However, the resulting behaviour is clearly undesirable as it ends up sending repeated messages.

Hall's view suggests the need for an approach to the feature interaction problem that can detect non-binary interactions. In such an approach feature interaction detection would involve an analysis of the compositional effects of two features relative to a *third requirement*. The purpose of the third requirement would be to prevent the undesirable behaviour resulting from the composition of the first two features from occurring. Note that this does not necessarily render invalid the framework proposed in Figure 3.1.

In his earlier work [Hall 2000a], Hall noted that combined properties resulting from such compositions are often "*.....inconsistent with natural individual feature correctness properties, leading to non-monotonic behaviour that requires design changes after feature*

combination." This suggests that the framework proposed in Figure 3.1 needs to be extended with an iterative approach. An iterative approach would be useful as a systematic way of eliciting the third requirement whose implementation would counteract the negative compositional effects. A challenge of such an approach is that the third requirement is not always explicit and known prior to observation of compositional effects.

Requirements Interaction Management: Robinson *et al.* [Robinson *et al.* 2003] proposed three properties of requirements interactions: *Basis*, *Degree* and *Direction*, and *Likelihood.* The *basis* specifies the basic elements of the feature interaction, that is, the minimum set of conditions that imply an interaction between features. This is similar to the five feature interaction types we have discussed above. The *degree* specifies the impact of the interaction on the operation of the system and the *direction* specifies whether the interaction is negative (undesired) or positive (desired) with respect to the satisfaction of system composition requirements. The degree and direction is a measure of the *interaction level* of a given set of features and may help in prioritising the resolution of undesirable feature interactions. Negative interactions with a high negative impact should be given a higher priority than those with a lower negative impact. The *likelihood* of a feature interaction determines its probability of occurrence.

These properties seem a valuable criterion for evaluating feature interactions. However, there are no systematic methods that put these criteria into practice. For example, there is no evidence of approaches for measuring *degree* and *likelihood* of feature interactions. Based on characteristics of feature interaction taxonomies, current approaches can only detect that an undesirable feature interaction may occur but can not tell us what the impact is (beyond that a requirement will be violated) and how likely it is that the feature interaction would occur.

3.1.4 Summary

We have argued that it is impossible to have feature interaction unless there is shared context between features and hence feature interaction is a context sharing problem. We have advanced this argument by illustrating the role of context in: (1) formalisation of feature interaction using the entailment relation, (2) documented sources of feature interactions, and (3) taxonomies of feature interactions. The characterisation of feature interactions through the taxonomies discussed in section 3.1.3 forms the basis for detection. In the next section we review current approaches to feature interaction detection, using design-time approaches.

3.2 Design-time Approaches

Typically, design-time approaches to addressing the feature interaction problem are based on formal methods. Formal methods are precise languages and techniques for specifying and analysing software systems. Due to their rigour, precision, and systematic treatment they are highly desirable in the development of software systems where a high standard of safety and integrity is essential [Yu and Dias 1993]. Such is the case with critical systems.

The application of formal methods in the detection of feature interactions involves describing feature behaviour using formal languages such as Temporal Logic [Felty and Namjoshi 2003]. The compositions of the feature behavioural descriptions are then analysed for conflicts by applying appropriate reasoning mechanisms such as model checking [Calder and Miller 2001]. In section 3.2.1 we review formal feature behavioural description languages and discuss corresponding conflict analysis mechanisms. Section 3.2.2 presents a summary and limitations of design-time approaches used in feature interaction detection.

3.2.1 Feature Behavioural Description Languages

Languages used for describing the behaviour of features in formal approaches addressing the feature interaction problem can be classified into *Logic Based*, *State-Based*, *Algebraic*, and

Structural. This classification is based on classifications proposed in Liu *et al.* [Liu *et al.* 1997], Turner *et al.* [Turner *et al.* 2004], and Kryvyi and Matveyeva [Kryvyi and Matveyeva 2003]. Following is a brief summary of the features of each formal language class.

Logic: This approach involves the use of logics to describe system desired properties. Validity of properties is checked using the associated axiom system of the used logic. The commonest type of logic systems used for specifying features and reasoning about their compositional behaviours are Modal Temporal Logic and the Event Calculus. Modal Temporal Logic expresses how the system behaviour evolves over time, making it possible to make statements about future states of the system. It can be used to reason about qualitative and quantitative temporal properties.

Qualitative properties include safety properties (such as mutual exclusion and absence of deadlocks) and liveness properties (such as termination and responsiveness). Examples of quantitative properties include periodicity, deadline, and delays. Temporal logic has been used for specifying the behaviour of telecommunications features with the model checking tool SPIN to automate the process of detecting interactions [Felty and Namjoshi 2003; Calder and Miller 2006].

As introduced in Chapter 2, the Event Calculus [Shanahan 1999] is a logical language for representing and reasoning about actions and their effects. It is also being used for specifying and analysing feature-based system behaviour [Laney *et al.* 2007]. An Event Calculus description relates *initiating* and *terminating* events to system states called *fluents*. A fluent is a property of the system that holds after it is initiated by an event and ceases to hold when terminated by another. An event e_1 is said to initiate a fluent f if upon occurrence of e_1, f becomes true. Meanwhile an event e_2 is said to terminate fluent f if its occurrence makes f false. This logic system has been used for analysing conflicts between policy specifications [Bandara *et al.* 2003], avoiding feature interactions resulting from inconsistent smart home

features [Laney *et al.* 2007], and real-time monitoring of requirements satisfaction in service-based systems [Spanoudakis and Mahbub 2006].

Yokogawa *et al.* [Yokogawa *et al.* 2003] proposed using bounded model checking for the detection of feature interactions. In this approach, the problem of feature interaction is reduced to that of the propositional satisfiability decision problem [Bordeaux *et al.* 2006]. The idea of propositional satisfiability is to determine if a specification exists that can satisfy a conjunction of requirements given some properties of the context. If no such specification exists, then the conjunction of the requirements is considered unsatisfiable in the given domain descriptions.

Mueller [Mueller 2006b] presents a comprehensive comparison between Event Calculus and Temporal Action Logic(TAL) [Gelfond and Lifschitz 1993] which could be useful as guidance in deciding which logic system to use for a given application. Giannapoulou and Magee [Giannakopoulou and Magee 2003] proposed an approach of translating event-based specifications into fluent propositions which makes them amenable to analysis with model-checking tools.

Features have also been specified as constraints on system behaviour and feature interactions defined as violation of such constraints. Accorsi *et al.* [Accorsi *et al.* 2000] proposed an approach in which features are specified as constraints and model checking tools are then used to analyse the specifications for feature interactions. In Elfe *et al.* [Elfe *et al.* 1998] a constraint-based approach for performing avoidance, detection, and resolution of feature interactions is proposed.

Hay and Atlee [Hay and Atlee 2000] proposed a transitions synchronisation technique called Conflict Free-Synchronisation. This technique allows features to simultaneously react to a

particular situation (such as a trigger event), but disables transition combinations that conflict. Two features conflict if the combination of their transitions violates relevant assertions.

State-Based: State-based languages are used to model the behaviour of a feature-based system in terms of abstract machines with sets of states and transitions between the states. The machine changes from one state to another depending on the input and may produce some output in response. Some notable examples of state-based languages include the Specification and Description Language (SDL) [Turner 2000; Kaindl 2005], and Message Sequence Charts (MSC) [Fu *et al.* 2000; Lorentsen *et al.* 2002; Uchitel and Chechik 2004; Damas *et al.* 2005].

The basic idea of feature interaction detection with state-based approaches is determining state reachability [Pomakis and Atlee 1996; Siddiqi and Atlee 2000a], i.e. whether all the states reachable in isolation are reachable in composition. A given state is associated with the satisfaction of certain properties; hence if the given state is not reached the satisfaction of these properties is violated.

SDL is an ITU z.100 standard language for analysing specifications for completeness and correctness, determining conformance of implementation to specifications, and determining consistency between specifications. It is intended for specification of complex, event-driven, real-time, and interactive applications which involve concurrent processes that communicate using discrete signals.

MSCs model system behaviour using scenario-based specifications and they focus on messages exchanged between features. A comprehensive survey of scenario-based notations in telecommunications systems development is documented in [Amyot and Eberlein 2003]. While it is relatively easy to communicate system functionality with scenarios, it is generally accepted that it is difficult to guarantee that complete system behaviour has been captured.

Algebraic: Algebraic approaches are similar to stated-based approaches. The only difference is that with algebraic approaches the consideration of state information is implicit and the focus is on actions that cause transitions between states. A feature-based system is modelled as a set of communicating processes with each process modelling a single feature. Each feature process describes the order in which events can occur (sequentially or concurrently).

An example of an Algebraic language is Language Of Temporal Ordering of Specification (LOTOS). In Fu *et al.* [Fu *et al.* 2000], LOTOS is used for describing feature specifications and these specifications are then translated into a state transition model that describes properties that should hold either globally or locally. These properties describe required feature behaviour and their violations are considered as feature interactions. State transitions that do not lead to property violations are encoded as Message Sequence Charts.

In [Gorse *et al.* 2006], a two-stage approach to detecting feature interactions in LOTOS specifications is proposed. The first stage is *filtering* in which possible interactions are detected by considering feature prestates, trigger events, post-conditions, and constraints. Nakamura *et al.* [Nakamura *et al.* 2000; Nakamura *et al.* 2002] proposed heuristics for filtering based on features specified with Use Case Maps.

The second stage is *testing*. At this stage suspect interactions identified in the filtering stage are further analysed to ascertain if they can actually occur. It is generally accepted that testing does not guarantee the absence of feature interactions [Godskesen 1995]. Since this approach is based on testing, it follows that it does not guarantee that all possible interactions have been detected.

Structural: With structural approaches the organisation of the system is defined in terms of its components – the features. Structural approaches are useful as visual notation for representing sequences of actions and the causality among them, e.g. Use Case Maps (UCM)

[Amyot 2001]. Structural approaches are not formal in themselves and consequently they are often accompanied by a formal underpinning which describes rules of valid connections between components. This is demonstrated in [Nakamura *et al.* 2000] where a formal link is provided from UCM to LOTOS.

Architecture-centric methods to handling feature interactions such as the DFC [Jackson and Zave 1998] and the Feature Stack Architecture [Pomakis and Atlee 1996] demonstrate the use of structural approaches. Both of these methods resolve non-deterministic feature behaviour by prioritising features, ensuring that they execute in a deterministic way. Practical application of the DFC (initially developed for Plain Old Telephone System (POTS)) has been demonstrated through its implementation in an IP telephony platform called BoxOS [Bond *et al.* 2004].

Petri Nets provide a graphical representation with formal semantics of system behaviour and they can deal with concurrency, non-determinism, and casual connections between events. In the approach proposed in Lu *et al.* [Lu *et al.* 2001], feature functionality is represented as a temporal formula and the behaviour of the featured-based system is represented as the set of all firing sequences. Feature interactions are detected by inspecting whether or not the temporal formula is violated when executing some of the firing sequences. The CHISEL notation [Turner 2000] is an informal graphical notation describing telecommunications features and services. Its graphical descriptions are supported by LOTOS and SDL.

Summary: Table 3.3 presents a summary of the combination of formal languages and reasoning mechanisms discussed above. For each category of formal language the table shows the specific notations used for feature behaviour description, the type of feature interaction detected, the feature interaction detection mechanism used, the application domain, and tool

support (where available). This table shows no evidence of approaches that address the detection of looping interactions.

Table 3.3. A summary of formal approaches

Approach	Feature Specification Notation (s)	Type of Interaction (s) Detected	Approach to Feature Interaction Detection	Application Domain(s)	Tool Support	References
State / Model-Based	Procedural Event-Based Formalism (P-EBF), Traces of Finite State Automata, MSC, and SDL	Negative Impact, and Bypass,	Constraint Satisfaction, Reachability Analysis, and Simulation	Intelligent Networks (IN) services, POTS and Distributed SIP Services.	ISAT LTSA [Giannakopoulou and Magee 2003]	[Hall 2000a], [Elfe *et al.* 1998], [Kaindl 2005], [Turner 2000], [Fu *et al.* 2000], [Damas *et al.* 2005], [Lorentsen *et al.* 2002], and [Uchitel and Chechik 2004]
Logic-Based	Linear Temporal Logic Formulas and Event Calculus Descriptions	Non-Determinism and Invariant Violation (Bypass)	Model Checking, Constraints Satisfaction, and Logic Deduction.	POTS, Smart Homes, Policies	SPIN and Event Calculus Planner	[Calder and Miller 2006], [Felty and Namjoshi 2003],[Laney *et al.* 2005],[Bandara *et al.* 2003], [Accorsi *et al.* 2000], [Shanahan 1999], and [Dini *et al.* 2004; Reiff-Marganiec 2004; Blair and Turner 2005]
Algebraic	LOTOS	Bypass, Override, and Negative Impact	Constraint Satisfaction	POTS	ELUDO, CADP, and LOLA [Fu *et al.* 2000]	[Fu *et al.* 2000], [Gorse *et al.* 2006], and [Nakamura *et al.* 2000; Nakamura *et al.* 2002],
Structural	Petri-nets and Use Case Maps (UCMs)	Non-determinism	Simulation and reachability analysis	POTS and User Interfaces for Mobile Phones	DESIGN/CPN [Albert *et al.* 1989]	[Amyot 2001], [Nakamura *et al.* 2000], [Jackson and Zave 1998], [Pomakis and Atlee 1996], [Kryvyi and Matveyeva 2003], [Lu *et al.* 2001], and [Lorentsen *et al.* 2002].

3.2.2 Limitations of Formal Approaches to Feature Interaction Detection

Formal specifications of features help improve clarity and precision [Calder and Miller 2006] in modelling feature behaviour. Formal analysis of feature compositions allows for rigour in the detection of feature interactions [Calder *et al.* 2003]. However, although the application of formal approaches has proven valuable in understanding feature interactions, especially in the telecommunications domain, the main challenges for their applicability concern end-user programming. The main goal of end-user programming is to equip end-users (rather than developers) with tools for designing and composing their features [Kolberg and Magill 2007]. This is different from current practice in which features are designed and composed by experienced developers.

Such a development paradigm raises two issues: (1) how can formal approaches be used to capture and formalise user intentions, and (2) how to handle feature interactions between end-user defined features. Some feature interactions can be traced to the way user intentions are interpreted [Stepien and Logrippo 1994; Xu *et al.* 2007]. Composition of features from different users will require user intentions to be captured and formalised and such information may aid accurate detection. For example, a looping interaction can be desirable or undesirable. Whether an interaction has a negative or positive impact depends on the composition requirement [Laney *et al.* 2007]. A composition requirement states what the desirable behaviour of the combination should be and is based on intentions of the composition. Hence, explicit knowledge of user intentions of the composition is important as it may minimise false detection of conflicts.

Formal approaches have so far been used for offline detection and resolution of feature interactions. Resolving feature interactions offline often involve re-specifying features such that the conflict is designed away [Hay and Atlee 2000; Calder and Miller 2006]. This implies customising the behaviour of features involved in a conflict to each other. Such customisation breaks the modularity of individual features [Hall 2005] and very often these resolutions are over-restrictive on the composition requirement [Laney *et al.* 2007]. Offline approaches are mostly suitable when the development of features and their composition is undertaken within constraints of well defined standards.

3.3 Runtime Approaches

The distribution of the development of features without well-defined standards among designers requires runtime approaches to detecting and resolving feature interactions. Such is the case with the internet telephony domain where users are able to create their own features [Nakamura *et al.* 2004b; Amyot *et al.* 2005; Wu and Schulzrinne 2007]. The basic idea for detecting feature interactions is to characterise them in terms of taxonomies such as those

discussed in section 3.1.3. This is analogous to the approach used in medicine for diagnosing diseases. Diseases are characterised in terms of their symptoms. These symptoms are then used to recognise an instance of the disease when it affects a particular patient. Similarly, characterising feature interactions in terms of taxonomies supports the detection of the feature interactions should they occur later, as demonstrated in Shehata *et al.* [Shehata *et al.* 2007b].

This idea for detection, as explained above, seems to apply in both design-time and runtime approaches. However, this is not true for resolution approaches. A major difference is that in resolving runtime feature interactions the option of redesigning features is not available. More importantly, at runtime, resolution has to be performed within relatively short time limits with minimal manual intervention. This implies the need for an approach that allows for generic resolutions techniques associated with known types of feature interactions. The resolutions can then be chosen at runtime when the corresponding feature interaction occurs. Offline approaches are not suitable for addressing this problem. In the following subsections we review two main classes of runtime approaches to feature interaction resolution: negotiation and arbitration.

3.3.1 Negotiation Approaches

Negotiation is a dialogue between features intended to find strategies to satisfy their requirements without causing a conflict [Velthuijsen 1993]. Each feature is implemented as a negotiating agent. The dialogue consists of proposals and counterproposals. In turn each proposal or counterproposal consists of strategies. A negotiation starts by one agent generating a proposal acceptable to it and sending the proposal to its counterpart (another agent). One receiving the proposal the second agent accesses it to determine if it is acceptable. A proposal is acceptable if it will lead to the satisfaction of the agent's requirements. If the proposal is not acceptable, a counterproposal is generated and sent to the originating agent. This dialogue continues until an acceptable proposal is agreed between the agents or it is

determined that it is impossible to have a proposal that can satisfy the requirements of both agents without a conflict.

Velthuijsen [Velthuijsen 1993] identified three configuration of negotiation schemes: *direct*, *indirect*, and *arbitrated* negotiation. In direct negotiation agents exchange proposals and counterproposals directly without a mediator. The multistage negotiation for distributed constraint satisfaction approach proposed in [Conry *et al.* 1991] is an example of a direct negotiation approach. Such direct dealings between agents have some disadvantages. (1) It increases the possibility that, in the course of resolving a conflict, they (agents) may reveal to each other confidential information. For example a subscriber to a Terminating Call Screening (TCS) may not want it revealed to a caller that a call has failed because the calling number is in their screening list. (2) There is a potential for deadlock if the agents fail to find a proposal they can both agree on.

With indirect negotiation, agents negotiate through a *negotiator* whose role include routing messages between agents, monitoring the progress of the dialogue, and suggesting proposals to the agents which may lead to a successful negotiation. Arbitrated negotiation is a form of indirect negotiation where the negotiator is an *arbitrator*. The arbitrator examines the requirements of the negotiating agents and imposes a binding proposal on how the conflict should be resolved.

There is no evidence in the literature which suggests that negotiation has been successfully applied to resolving types of feature interactions other than non-determinism. This does not necessarily mean that negotiation is limited to resolving non-deterministic feature interactions. For non-deterministic conflict, negotiation is about reaching an agreement on which feature can have access of the contested resource. Therefore resolving non-deterministic feature interactions using negotiation means reaching a mutually agreed prioritisation between the features agents.

3.3.2 Arbitration Approaches

Current approaches to managing feature interactions at runtime are based on the concept of *arbitration*. As stated in chapter 1, arbitration is a legal technique for dispute resolution outside courts, in which the parties to a dispute refer to one or more persons (the arbitrators), whose decision on how the dispute should be resolved is binding [Bonn 1972]. Feature *specifications* satisfy associated *requirements* by issuing actions which effect changes on the shared *context* [Jackson 2001]. An *arbitrator* is placed between feature specifications and the real world context they interact with to satisfy their requirements, and hence intercedes between feature specifications and shared resources. This is illustrated in Figure 3.5. S_1 and S_2 are specifications which satisfy requirements R_1 and R_2, respectively. The specifications satisfy their requirements by interacting with the shared resource (W_s).

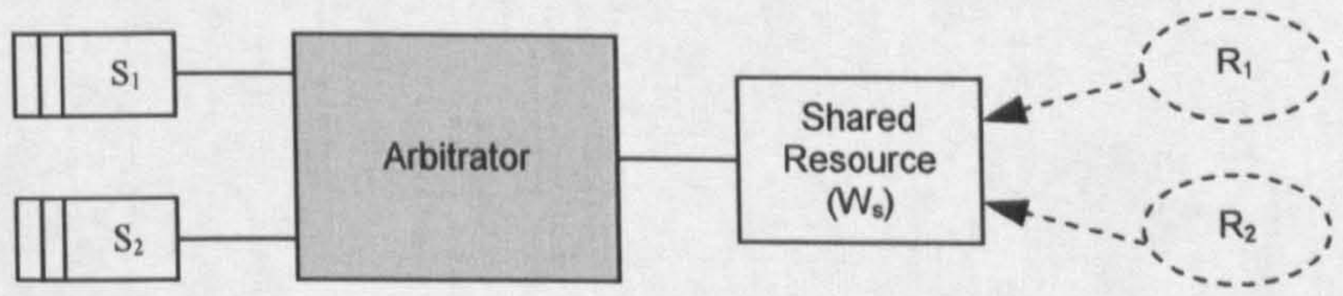


Figure 3.5 Generic Composition of Features through an Arbitrator

Actions issued according to the specifications have to be approved by the arbitrator before they can be passed on to the shared resource. In this section we review three arbitration approaches: the Feature Interaction Manager (FIM) [Tsang and Magill 1998], Composition Controller [Laney *et al.* 2007], and the Modular Supervisory Control with Priorities (MSCP) [Chen *et al.* 1995].

The Feature Interaction Manager Approach: A Feature Interaction Manager has two modes of operation: *learning* and *management*. In learning mode each feature is executed in a test environment and its external behaviour, viewed in terms of event sequences, is recorded as the feature's *behavioural signature*. In management mode *actual feature behaviour* is

compared against the previously recorded behavioural signatures. Deviation of actual feature behaviour from recorded behaviour is reported as a feature interaction. Detected feature interactions are resolved through *error recovery* and *prioritisation* techniques.

Figure 3.6 shows a FIM in the context of a telecommunications switching system. A Service Logic Program instance (*SLPi*) implements a single feature. The FIM is positioned between the Basic Call Manager (BCM) and the *SLPis* such that information flowing in and out of an SLPi passes through the FIM. SLPi input information include trigger events and network resource status, while output information include resource manipulation requests.

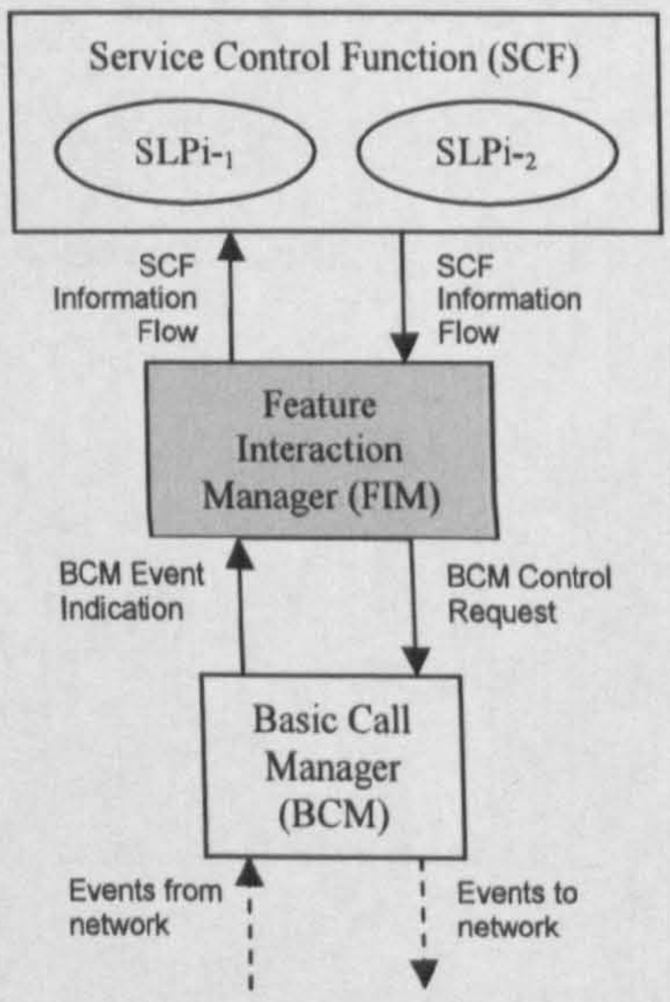


Figure 3.6 Conceptual FIM Approach (Adapted from [Tsang and Magill 1998], p 824).

The capture of feature behaviour in terms of behavioural signatures facilitates flexibility in the approach as conflicts can be detected without prior knowledge of the requirements of a feature. However, the learning phase introduces an overhead and limits the application of the approach when features need to be composed dynamically at run-time. In studies reported in [Tsang and Magill 1997], the use of signatures was demonstrated to lead inaccurate detection of feature interactions since not all deviations are necessarily conflicts.

In the FIM conflicts are allowed to occur and error recovery techniques (such as roll-back) are used to bring the system to a consistent state. Roll-back techniques are not appropriate for systems where satisfaction of the requirement include physical changes. With the exception of recent work by [Kolberg *et al.* 2001; Kolberg *et al.* 2003; Kolberg and Magill 2007], the feature interaction problem has so far been studied in the context of violation of requirements where such violation does not lead to a physical damage. For example if the occurrence of a feature interaction in a telecommunications switching system leads to the disconnection of a voice call, the connection can be restored (re-initialised) with a fresh call attempt. On the contrary, when satisfying a requirement requires a physical change then the effects of violation by a feature interaction may be irreversible. For example, in an automobile, if a feature interaction results in failure of brakes and the car hits a pedestrian, roll-back cannot reverse the resulting effects.

The Composition Controller Approach: The Composition Controller (CC) approach addresses some of the limitations of the FIM approach. As introduced in Chapter 2, the CC approach is based on the Problem Frames approach to software development. It generalises the FIM approach by considering an SLPi as a *specification* of a feature and the BCM as a shared *problem domain*. In this approach feature interaction detection and prior knowledge of feature behaviour are not necessary. This is because the CC approach assumes that feature specifications do not only specify the events that should occur for a requirement to be satisfied but they also specify events whose occurrence may violate the requirement. In this approach a feature interaction occurs if one feature issues an event that is currently prohibited by another feature.

Violating events are specified in *prohibit(...)* clauses which instruct the composition controller to disallow events that have a potential to violate requirements. These events are prevented from passing to the shared domains within a given period of time to allow satisfaction of the requirement to be upheld. Unlike in the FIM approach, through event

prohibition, the CC approach minimises the possibility of a domain being in an inconsistent state, hence there is no need for error recovery. In essence the CC approach is *proactive* in the sense that it *prevents* the occurrence of conflicts. In contrast the FIM is *reactive* in that it allows conflicts to occur and then take measures to recover from the consequences.

Modular Supervisory Control Approach: The modular supervisory control with priorities (MSCP) [Chen *et al.* 1995] approach for discrete event systems arbitrates between supervisors controlling the same process (or plant). The approach is based on Brandin and Wonham's Supervisory Control Theory for discrete event systems [Brandin and Wonham 1994]. Given a *process* the objective of this theory is to design a *supervisor* in such a way that the process coupled with the supervisor behaves according to various *constraints* [Charbonnier *et al.* 1999]. The conceptual structuring in this approach is similar to Problem Frames [Jackson 2001]. The process is the *problem domain*, the constraints are the *requirements*, and the supervisor is the *specification*.

In Chen *et al.* [Chen *et al.* 1995], the importance of addressing initialisation concerns is recognised. They recognise that to ensure that when a higher priority feature pre-empts a lower priority feature, the lower priority feature is able to resume its control on the resource correctly, each feature must be equipped with mechanisms to keep track of the current state of the resource. This addresses part of the problem as mechanisms to monitor state changes on the shared domain are part of the initialisation problem. The other part of the problem is that on resumption the feature needs a way of engaging the correct behaviour in order to satisfy its requirement. However, similar to other run-time approaches [Tsang and Magill 1997; Hay and Atlee 2000; Blair *et al.* 2002; Laney *et al.* 2005], the MSCP approach defers the latter part of the initialisation problem to the solution space. This limits the scope of solutions available to resolve a conflict [Tsang and Magill 1998; Jia and Atlee 2004].

In the conflict resolution scheme proposed in [Wong *et al.* 2000], a suspended feature can resume control only when the states of the shared resource and other supervisors have (voluntarily) returned to what they were at the point of suspension. This scheme ensures that a suspended feature can resume control only when the shared resource is in a safe state. However, this also implies that if the resource has not been returned to the safe state then the suspended feature will wait for its chance indefinitely even if the resource is free.

This suggests a need for an approach for analysing initialisation concerns in the problem space and derivation of appropriate solutions which can then be used to prevent conflicts at runtime. Such an approach could widen the scope of resolutions to conflicts and guarantee that the requirements of the conflicting features are eventually satisfied. We conclude this section by presenting a comparative summary of the three arbitration approaches discussed above.

Comparative Summary of Arbitration Approaches: Table 3.4 presents a comparative summary of the characteristics of the FIM, CC, MSCP approaches. The main strength of the FIM over the CC is that it treats feature specifications as 'black-boxes' hence does not dictate any changes in the way they are specified. This makes it suitable for use in legacy system and environments where features are developed by different designers and as such only the externally observable feature behaviour is available. Its main drawback is that errors are allowed to occur and such errors may potentially take the system to an inconsistent state.

Error recovery techniques are used to bring the system to a consistent state. While using error recovery techniques is plausible for telecommunications switching systems, allowing errors to occur may have irreversible consequences for systems that rely on the physical environment to satisfy their requirements. For example in an automobile system, if the requirement is that brakes should be applied to avoid hitting a pedestrian, error recovery cannot correct the consequences of the violation of such a requirement.

Table 3.4 Comparative Summary of Characteristics of Arbitration Approaches

	Arbitration Approaches		
Comparison Criteria	**Feature Interaction Manager (FIM)**	**Composition Controller (CC)**	**Modular Supervisory Control with Priorities (MSCP)**
Method of Dealing with Shared Domain Inconsistency	POST-ACTION (REACTIVE): Errors are allowed to occur and error recovery techniques are used to bring the feature-based application to a consistent state.	PRE-ACTION (PROACTIVE): Prohibit clauses prevent potentially violating events from taking effect on the shared domain states.	PROACTIVE: Events with higher priority supervisors are allowed to temporarily override those of lower priority in case of potential blocking.
Method of Feature Interaction detection	Detection is through the identification of deviation from previously stored signature behaviour.	Detection is through the occurrence of a currently prohibited event.	Detection is through explicit specification of "out-of-specification" states which indicate violation of individual supervisor constraints.
Method of Feature Interaction Resolution	Error Recovery and feature prioritisation	Feature prioritisation	Feature prioritisation based on event priority functions.
Support for the eventual satisfaction of requirements of a feature that is granted access to a resource	NONE	NONE	NONE
Explicit support for initialisation of shared domain	NONE	NONE	NONE
Support for dynamically changing feature sets.	YES: Through dynamic loading and unloading of feature behavioural signatures into signature database.	YES: Feature specifications are composed through a composition controller.	YES: Supervisors are dynamically composed through a coordinator.
Require changes in the way features are specified?	NO: Management of feature interactions is based on externally observed feature behaviour.	YES: requires that specifications include prohibited events.	YES: Assumes that specifications are event-based with their behaviour modelled as state machines.

On the one hand the Composition Controller, through event prohibition, disallows events that are likely to lead to the violation of the requirement of a currently executing feature. However, this approach assumes that feature specifications issue *prohibit(...)* events which are used to detect and resolve potential feature interactions. This assumption limits the application of this approach as it requires changes to the way features are specified.

MSCP is similar to CC in that to prevent blocking on shared resources, events of higher priority features are allowed to temporarily override those of lower priority. The key difference is that in the MSCP approach the events are rejected based only on the priority between features. These may result in all events of a lower priority feature being rejected; even those that will not cause blocking (conflict). Meanwhile in the CC, events to be rejected by the arbitrator are explicitly described in the individual feature specifications. This enables the arbitrator to reject only the specific events that are likely to cause a conflict rather than the entire set of events for a lower priority feature. Hence in the CC, in addition to priority between features, events rejection is also based on specific conflict prone events described in the specification.

3.4 Chapter Summary

In this chapter we have explored the argument that the feature interaction problem is a context sharing problem and reviewed approaches to handling feature interactions both at design-time and at run-time. We have noted that design-time approaches are not suitable in the resolution of conflicts at runtime since they often require redesign of features and are often over-restrictive on the composition requirement. Postponing the resolution of feature interactions to runtime has the advantage of resolving actual rather than potential interactions. We have reviewed two runtime approaches to resolving feature interactions: negotiation and arbitration.

In a negotiation approach features are designed as negotiating agents. In the event of a conflict they enter into a dialogue on how the conflict can be resolved. Such a dialogue results in one of the features being granted access to a shared resource. With arbitration, a third party component presides over the conflict and its decision on how the conflict should be resolved is binding on the features involved. Prioritisation is a common resolution technique [Hay and Atlee 2000],[Jackson and Zave 1998]. Both negotiation and arbitration resolve non-

determinism conflicts using prioritisation. There are differences though. In negotiation prioritisation is implicit as it is reached through a dialogue and a resolution is not guaranteed. The lack of a guarantee for a resolution is due to the potential that a dialogue may reach a deadlock. Meanwhile, in arbitration prioritisation is explicit and a resolution is guaranteed.

Prioritisation ensures that in the event of a conflict a higher-priority feature is given control of the shared resource. However, this conflict resolution technique does not guarantee that the requirement of the feature that eventually gains control of the resource will be satisfied (as shown in the comparative summary in Table 3.4). When a previously pre-empted feature has to resume execution it has to be initialised correctly with the shared resource. This is because its model of the shared resource may be inconsistent with the actual state of the resource, due to the state being changed by another feature. This may result in its requirement not being satisfied when it is eventually granted access to the resource. We have characterised this as the initialisation problem. Arbitration and negotiation approaches do not address this problem.

In summary, arbitration and negotiation only address the resolution of non-deterministic and negative impact feature interactions. They are not sufficient for runtime resolution of *bypass*, *invocation order*, and *looping interactions*. In this thesis we propose an approach to resolving runtime feature interactions which extend arbitration with a technique to resolve bypass feature interactions. Bypass interactions result from inability of current approaches to address initialisation concerns. Therefore our approach to runtime resolution of bypass interactions involves a solution to the initialisation problem.

Chapter 4. Complementing Arbitration with Contingency Planning

Feature interactions may occur whenever features are composed such that they control a shared resource. Features with inconsistent requirements may prevent, block, or interfere with the satisfaction of the requirements they satisfied in isolation. Composing such features can be complex. This is because the model of the shared resource in one feature may become inconsistent with the actual state (of the shared resource), due to the state being changed by another feature. This may result in the requirements of one of the features not being satisfied. In Chapter 1 we characterised this as the *initialisation problem*. We noted that arbitration alone is insufficient in addressing this problem.

Our approach to addressing the initialisation problem is based on *contingency planning*. We use contingency planning as a mechanism for identifying varying conditions and corresponding alternative behaviours necessary to guarantee that when an application begins execution it is able to satisfy its requirement by adjusting its behaviour to the state of the resource. This is achieved by equipping each feature with contingent specifications corresponding to each state of the shared resource. Depending on the current state, one of the contingencies is selected to enable a feature to satisfy its requirement.

Contingency planning ensures that in the event of a non-deterministic conflict the requirements of conflicting features are eventually satisfied. Our approach ensures that all features have a consistent view of the state of the shared resource and are able to use the shared resource at any state. In this short chapter we outline the conceptual basis for our approach by motivating how the concept of contingency planning is relevant to feature interaction resolution. We also argue that the concepts of arbitration and contingency are

complementary as their combination ensures that in the event of a non-deterministic conflict the requirements of conflicting features are eventually satisfied.

In section 4.1, we revisit the initialisation problem explaining it in more technical depth than presented earlier. Domain descriptions are essential in addressing initialisation problems. We explain, justify, and illustrate their role in section 4.2. The application of the concept of contingency planning in our approach to addressing the initialisation problem is outlined in section 4.3.

For a feature to select the correct contingency for a given state of the context it is necessary to correctly determine the current state of a shared resource. We present a mechanism for detecting the current state in section 4.4. In section 4.5 we analytically validate our approach to the initialisation problem by comparing and contrasting its main features with related works addressing context switching and self-stabilisation problems in operating systems. Finally, in section 4.6 we conclude with a chapter summary.

4.1 The Initialisation Problem - Revisited

The need for initialisation arises whenever two or more features share the control of a resource. Before presenting our proposed approach to the initialisation problem we first recap and explain this problem in more detail than what we presented in chapter 1. Figure 4.1 shows a conceptual model of how an arbitrator resolves conflicts between two machines controlling a shared resource. To illustrate the initialisation problem consider that: machine M_1 issues a sequence of events $S_{m1} = \{e_{11}, e_{12}, ..., e_{1n}\}$ to satisfy requirement R_1; machine M_2 issues a sequence of events $S_{m2} = \{e_{21}, e_{22}, ..., e_{2m}\}$ to satisfy requirement R_2; and in response to the events issued by M_1 and M_2, the shared domain, W_s, changes between the sequence of states $\beta = \{\beta_1, \beta_2, ..., \beta_k\}$.

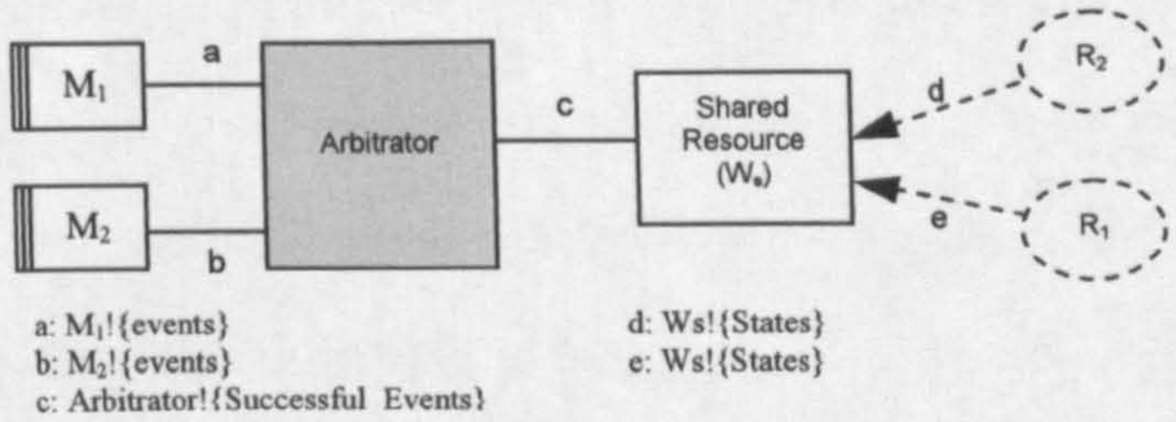


Figure 4.1 Composition of two machines with an arbitrator

The arbitrator intercedes between the machines and the shared domain. It filters events passed to W_s depending on its composition requirement which states the priority between R_1 and R_2. Both machines are allowed to effect state changes in W_s unless there is a conflict, in which case the machine corresponding to the requirement with the highest priority is the one whose events are allowed to effect state changes in W_s.

Assume that R_1 has a higher priority than R_2 and hence certain events issued by M_2 will be rejected at times when R_1 needs to be satisfied. Consider a scenario in which while M_2 is executing M_1 requests access to W_s. Since R_1 has a higher priority, M_1 pre-empts M_2. This means that all events from M_2 that conflict with the satisfaction of R_1 are rejected. Assume that the sequence of events issued by M_1 leaves the shared domain in some arbitrary state β_x, and that M_2 assumes W_s to be in some state β_y, where $x \neq y$.

When M_1 has finished executing, events from M_2 are no longer rejected. However, R_2 may not be satisfied since the actual state of W_s is not the same as that assumed by M_2. This is due to the interference of M_1 in changing the state of W_s. The initialisation problem is how to reconcile the state assumed by M_2 with the *actual state* of W_s to minimise the possibility of a discrepancy in the execution of M_2. In the rest of the thesis *Next machine* refers to the machine about to take control of the shared domain. *Current machine* refers to the machine that is currently interacting with the shared domain.

4.2 The Need for Domain Descriptions in Addressing Initialisation

A *domain description* describes the indicative behavioural properties of the context. It maps event occurrences to state changes and thus helps in reasoning about how successful events result in state changes in the context. An event is successful if it has not been rejected by the arbitrator. Addressing the initialisation problem requires explicit knowledge about behaviour of the context, the possible states, and what actions may lead to those states. Domain descriptions are about this information and hence they are essential in addressing initialisation concerns.

A useful analogy of a domain description is an interactive map of a country. A map shows the different cities, their relative position to each other, and the highways that connect them. Such information enables a person navigating the country to answer questions such as: If I am in city X, what motorway should I take to get to city Y? In software development, we use domain descriptions in a similar manner in describing and reasoning about the behaviour of resources.

Figure 1.1 is a behavioural model of the DVD-R. According to Figure 1.1, if the current state of the DVD-R is *Stopped*, the occurrence of a *play* event would result in the state changing to *Playing*. Machines interacting with the DVD-R issue events from the set *{play, record, stop, pause}* to satisfy their requirements. In response to these events, the DVD-R may be in one of the states: *{Playing, Recording, Stopped, Paused_Recording, Paused_Playing}*. In the next section we discuss how our approach applies the concept of contingency planning in preparing feature specifications for satisfying their requirements for different states of a shared resource based on domain descriptions.

4.3 Contingency Planning as an Approach to Addressing Initialisation Problems

A specification that assumes a single initial state of the shared resource may not satisfy its requirement. This is because for a shared resource, a single initial state can not be guaranteed to be true always. This is due to the possibility of machines of other features changing the state. Such interference may lead to inconsistency between the actual state of the domain and the state assumed by the next machine. We have characterised this as the initialisation problem.

In addressing the initialisation problem we propose the use of *contingency planning*. Contingency planning is a concept from management science [Umanath 2003; Sousa and Voss 2008]. In management, contingency planning entails explicit *a priori* statements about various situations which are not certain to happen but are nevertheless possible in the operations of an organisation. These situations are not part of the normal operations of the organisation and they are regarded as disruptions (or exceptions). Contingency planning is used as a risk management strategy aimed at designing corresponding alternatives for how the satisfaction of organisational goals will be maintained, should those situations arise.

In this thesis we use the term contingency to mean having several specifications per feature, satisfying the same requirement, depending on the current state of the shared resource. We refer to these as *contingent specifications*. Using contingency planning, each feature is equipped with contingent specifications corresponding to all possible states of the shared resource. Contingent specifications satisfy the same requirement and are selected depending on the current state of a shared resource. The derivation of contingent specifications involves identifying the set of all possible states of the shared resource from its behavioural description. In illustrating how we apply the concept of contingency, consider a resource with the set of states $\beta = \{\beta_1, \beta_2, \ldots, \beta_k\}$. Using the entailment relation, we express the structure of a feature as:

$$S, W \vdash R \qquad (4)$$

Based on the above expression we express the contingencies of a feature as follows:

$$S_1, W[\beta_1] \vdash R \qquad (4.1)$$

$$S_2, W[\beta_2] \vdash R \qquad (4.2)$$

.....

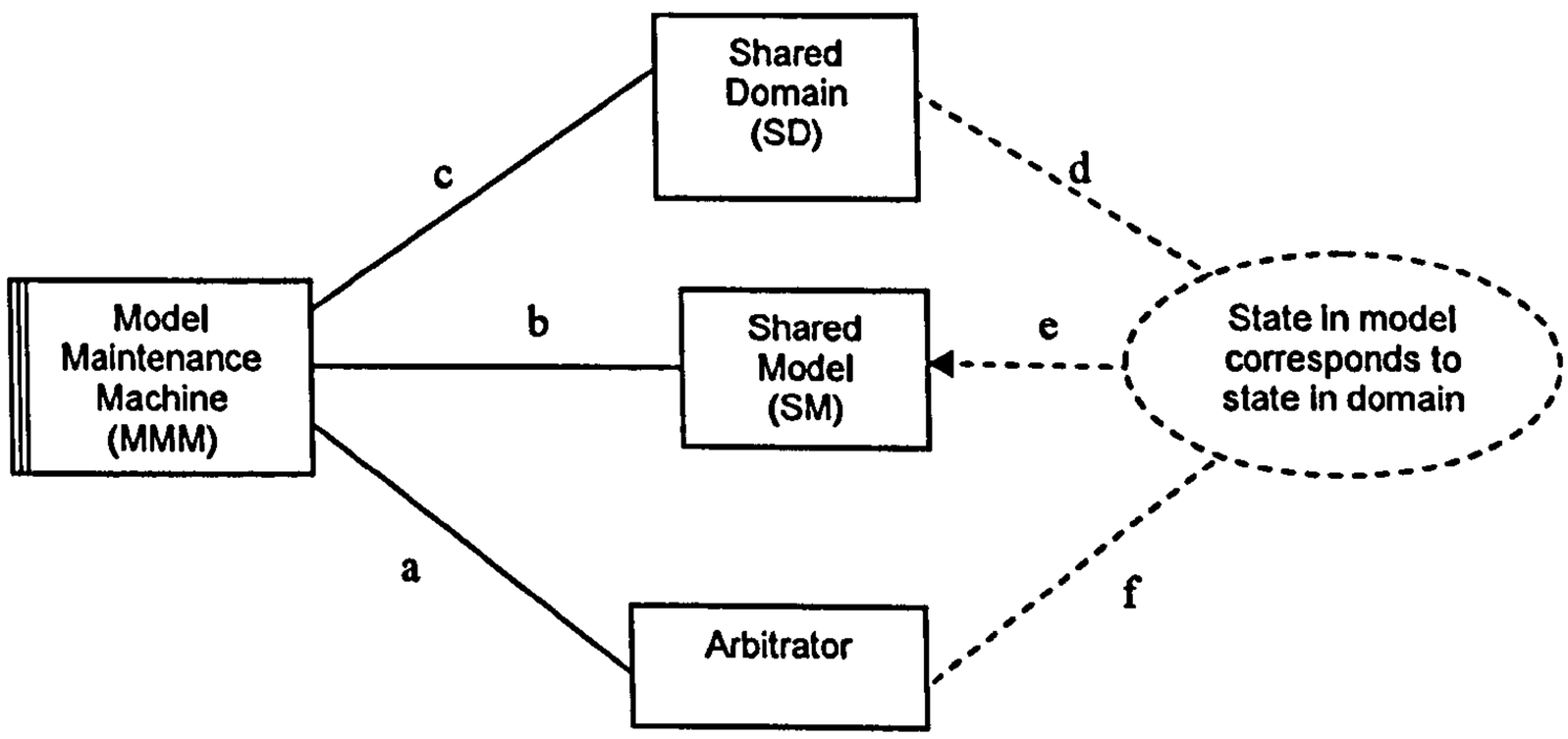
$$S_k, W[\beta_k] \vdash R \qquad (4.k)$$

Each of the relations 4.1 to 4.k are contingencies corresponding to each element in β. S_1 to S_k are contingent specifications corresponding to each of the states of the context shown inside the square brackets. Note that the requirement (R) and context (W) are the same in all the relations (4.1) to (4.k). What differs is the specifications (S) and the state of the context considered. Deriving a contingent specification involves answering the question: if the current state of the shared resource in *context* (W) is $\beta_x \in \beta$, what *specification* would satisfy the *requirement (R)*?

4.4 Determining the Current State of a Shared Resource

In order for each feature to select the correct contingent specification at run-time, we provide a mechanism for maintaining consistent value of the current state of a shared resource. This mechanism involves *creating and maintaining* a model of the resource shared by all features. The shared model acts as a reference for a feature to track state changes. The state in the model is updated by keeping track of all successful events. An event is successful if it has been allowed by the arbitrator. The model is based on a resource's behavioural domain descriptions and it enables a feature to determine the current state of a shared resource based on successful events. For example, when a *record* event occurs, a DVD-R starts *recording*. Once the model is created and synchronised with the actual state of the shared domain, future states of the resource can be determined by querying the model.

The assumption made in the state tracking mechanism is that the behaviour of a model accurately mimics that of the actual domain. The rationale behind this assumption is that the occurrence of a successful event results in corresponding state changes in the actual resource and its model. Figure 4.2 shows a problem diagram for creating and maintaining local models in each feature. The model maintenance machine ensures that the shared domain model corresponds to the actual state of the shared resource.




```
a: Arbitrator!{Successful_Events}
b: MMM!{EventsToModel}
c: MMM!{EventsToResource}
d: SD!{ResourceState}
e: SM!{ModelState}
f: Arbitrator!{Successful_Events}
```

Figure 4.2 Creating and Maintaining Model Subproblem (Adapted from [Jackson 2001])

In order to maintain an accurate model of the current state of the resource, the model maintenance machine monitors all successful events. In Figure 4.2 interface *a* represents successful events from the arbitrator. Successful events are forwarded to the shared resource through interface *c* and interface *d* represents the current state of the resource resulting from the events received through *c*. The shared model is updated through interface b and its current state is observed in interface *e*.

Figure 4.3 shows a problem diagram of the generic arbitrator of Figure 4.1 with the model maintenance mechanism and share model. Interfaces *a* and *b* represent the set of events issued according to the specifications of machines M_1 and M_2, respectively. If there is a conflict between the events issued by the two machines, the arbitrator rejects the event originating from a lower priority machine in favour of a higher priority machine. The successful event is then passed by the arbitrator to the model maintenance machine through interface *c*. The successful events are forwarded to the resource through interface *d* and its model is updated through interface *e*.

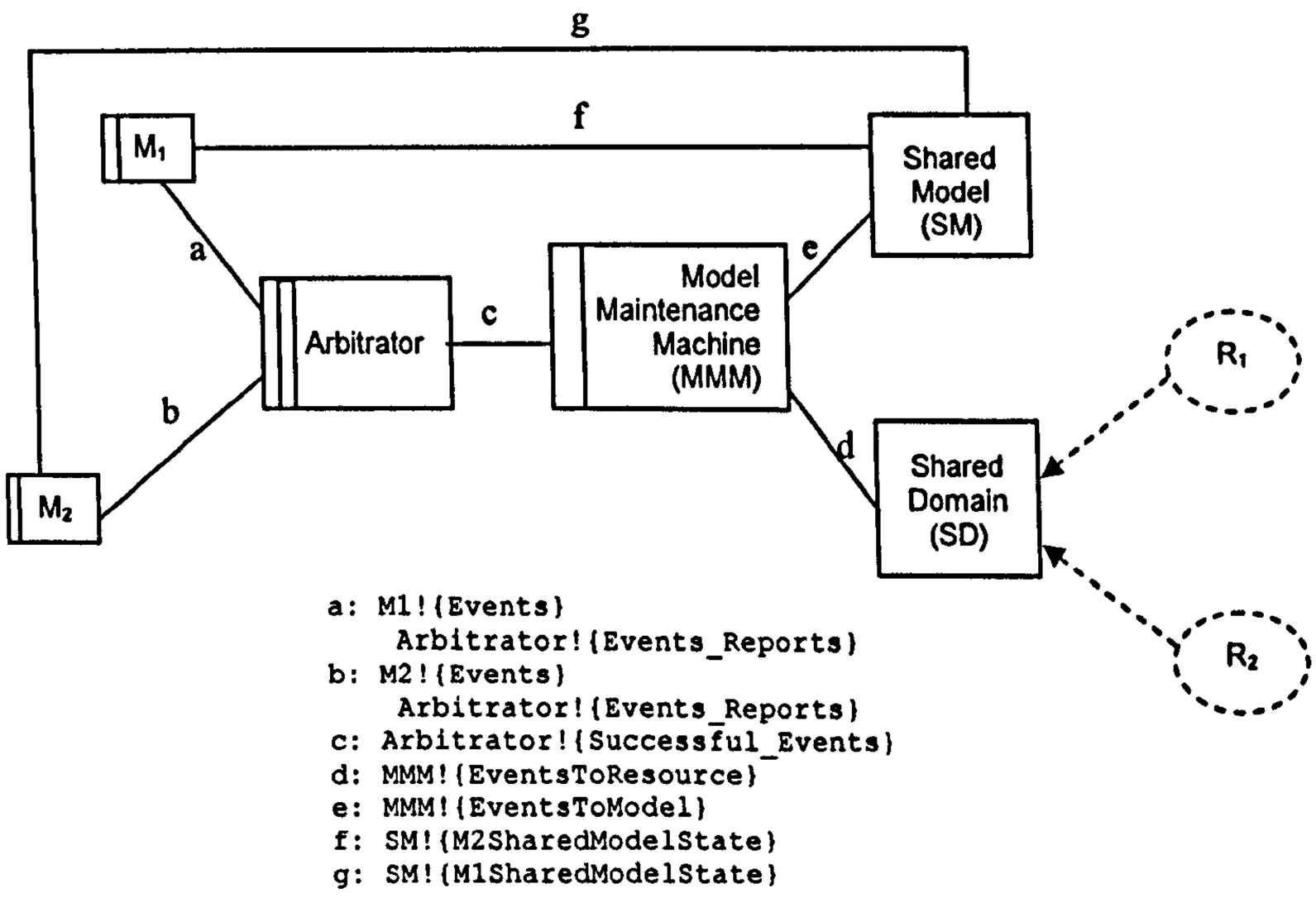


Figure 4.3 Arbitrator with Shared Resource State Tracking Mechanism

Machines M1 and M1 determine the current state of the shared resource through interfaces *f* and *g*, respectively. This mechanism ensures that a feature that is not currently in control of a shared resource can passively follow state changes, so that when it is granted access to the resource, it selects a suitable contingent specification.

4.5 Previous Attempts at Addressing Initialisation Problems

The initialisation problem (as characterised in this thesis) is similar to the problems of *context switching* [Silberschatz *et al.* 2004] and *self-stabilisation* [Dolev and Yagel 2008] in operating systems. In this section we compare such approaches to our proposed approach to solving the initialisation problem. Due to the similarity between the concepts of contingent specification and *context-awareness*, we also compare our approach to approaches used in developing context-aware applications.

Context Switching in Operating Systems: The initialisation problem is similar to the *context switching problem* in operating systems [Silberschatz *et al.* 2004]. The idea of context switching is to enable multiple applications to share the same processor, accessing it at specially allocated times called time slices. At each time slice, an application may execute a portion of its instruction set. When its time slice finishes, the application is suspended. When an application uses the processor, it changes the processor state such as values of registers and instruction counters. This may result in an application that was suspended being unable to resume correctly. Context switching is an approach to addressing this problem.

A context switcher saves the values of the registers when an application is suspended [Nuth and Dally 1991; Hwu and Conte 1994]. It reloads these values when the application resumes - ensuring that a suspended application starts correctly. In essence, the approach to the context switching problem is to adjust the state of the shared resource (processor) to that of the next machine. The assumption this approach makes is that it is possible to adjust the state of the resource to that expected by the application. However, this assumption does not always hold.

The focus in the context switching approach is on the state of the machine rather than the state of the world. It does not matter what state the world is in; what matters is what state is expected by the machine. In contrast, our approach focuses on the state of the resource and its relation to the satisfaction of the machine's requirement. In context switching, a suspended

machine resumes execution from the last instruction it executed before suspension. This inflexibility does not take into account that changes to the state of the resource that occurred while the machine was suspended may already be satisfying (or advanced towards satisfying) the machine's requirement.

As an illustration consider a machine with a requirement of closing a door. If the machine assumes that the door is initially open, if the door has already been closed by another machine there is no need to open only to close it again. Our approach works around this problem. Each contingent specification consists of actions that satisfy the requirement from a particular state of the resource. Each specification is executed only when the corresponding assumed initial state is the current state of the resource. This prevents the possibility of (unnecessarily) repeating actions already performed on the resource, and is similar to Scalera and Vazquez's technique [Scalera and Vazquez 1998] for minimising context switches by sharing data between contexts.

Self-Stabilising Operating Systems: Transient faults, such as soft errors (electrical spikes on the hardware) [Mitra *et al.* 2007], often result in operating systems being in arbitrary and unexpected states where it may be impossible to recover to normal operation [Dolev and Haviv 2006]. If such a fault occurs in an operating system controlling a satellite in space, that satellite may be lost. In Dolev and Yagel (2008) [Dolev and Yagel 2008], a self-stabilising approach to recovering from such faults is proposed. A system is said to be self-stabilising if it can be started in any possible initial state and converge to a desired behaviour.

The approach of self-stabilisation is to reload and re-execute the operating system code each time a Non-Maskable Interrupt (NMI) occurs [Dolev and Haviv 2006]. An NMI is used to signal the occurrence of hardware errors from which it is impossible to recover. The reload and re-execution of the operating system code ensures that in case the system ends-up in an unexpected state it can recover itself to normal operation. Although this approach offers a

generic solution to recovery from an arbitrary unexpected state, it does not take into account that the state reached as a result of a transient fault may be a desirable state with respect to satisfying the requirement.

In contrast, our approach considers the entire state space of a shared resource and devises plans on how satisfaction of the requirement will be achieved should the resource be in that state. The tailored recovery plans for each possible state eliminates the need for reload of the entire feature specification as, only the contingent specification corresponding to the current state needs to be executed.

Approaches to Developing Context-Aware Applications: Our approach to deriving contingent specifications is broadly related to work on developing context-aware applications [Oreizy *et al.* 1999; Cortellessa *et al.* 2000; McKinley *et al.* 2004; Zhang and Cheng 2005; Zhang *et al.* 2005a; Salifu *et al.* 2007]. While the emphasis of such work is on how to make a system context-aware, the focus of our work is how to use context-awareness techniques to manage inconsistency. Approaches to developing context-aware applications consider the entire space of possible changes in the context and design behaviours that enable the satisfaction of the application's requirement in those states. Although we also initially consider the entire state space of the context, at composition-time, this is reduced to only behaviours corresponding to states of the context that are possible. The set of likely states is determined by the requirements of the features being composed. The selection of contingent specification for composition is covered in chapter 5.

4.6 Chapter Summary

This chapter has outlined the conceptual basis of our approach to the initialisation problem by motivating how the concept of contingency planning is relevant to feature interaction resolution. Contingency planning enables features to deal with initialisation concerns. This is achieved by equipping each feature with contingent specifications corresponding to each state

of the shared resource. Depending on the current state, one of the contingencies is selected to enable a feature to satisfy its requirement.

We have discussed two main components of our solution: contingency analysis and determination of the current state of the resource. Contingency analysis involves identifying all possible states of the shared context and deriving corresponding contingent specifications. For an application to select a suitable contingency, it needs to determine the current state of the context. This is achieved by having a shared model of the dynamic behaviour of the shared resource. This model is updated by taking note of all successful events and their possible effect on the actual context. This enables the feature to determine the state of the context by querying the model.

We have also discussed the similarities and differences between our approach to the initialisation problem and approaches to context switching, and self-stabilisation in operating systems. The initialisation problem is very similar to the operating system context switching problem as they both involve ensuring that the requirement of a previously suspended application is satisfied when it resumes execution. The main difference is that in our approach to the initialisation problem a feature adjusts its assumed state to that of the context (obeys the world), meanwhile in context switching the reverse is true. The objective of self-stabilisation approaches is to ensure that an operating system is able to recover itself if it gets to an unexpected state.

In the next chapter we discuss how our approach can be instantiated using the problem frames notation to model features and the Event Calculus for domain descriptions.

Chapter 5. Using Contingency Planning and Arbitration to Resolve Runtime Conflicts

In chapter 4 we discussed the conceptual basis of our proposed approach to addressing the initialisation problem. We argued that contingency planning complements arbitration by ensuring that the requirement of a feature that eventually gains access to a shared resource is satisfied. This section illustrates how our proposed approach can be used in practice by showing how an existing arbitration approach can be extended with contingencies. We present the steps involved in developing a feature-based application that makes use of the two concepts to resolve feature interactions.

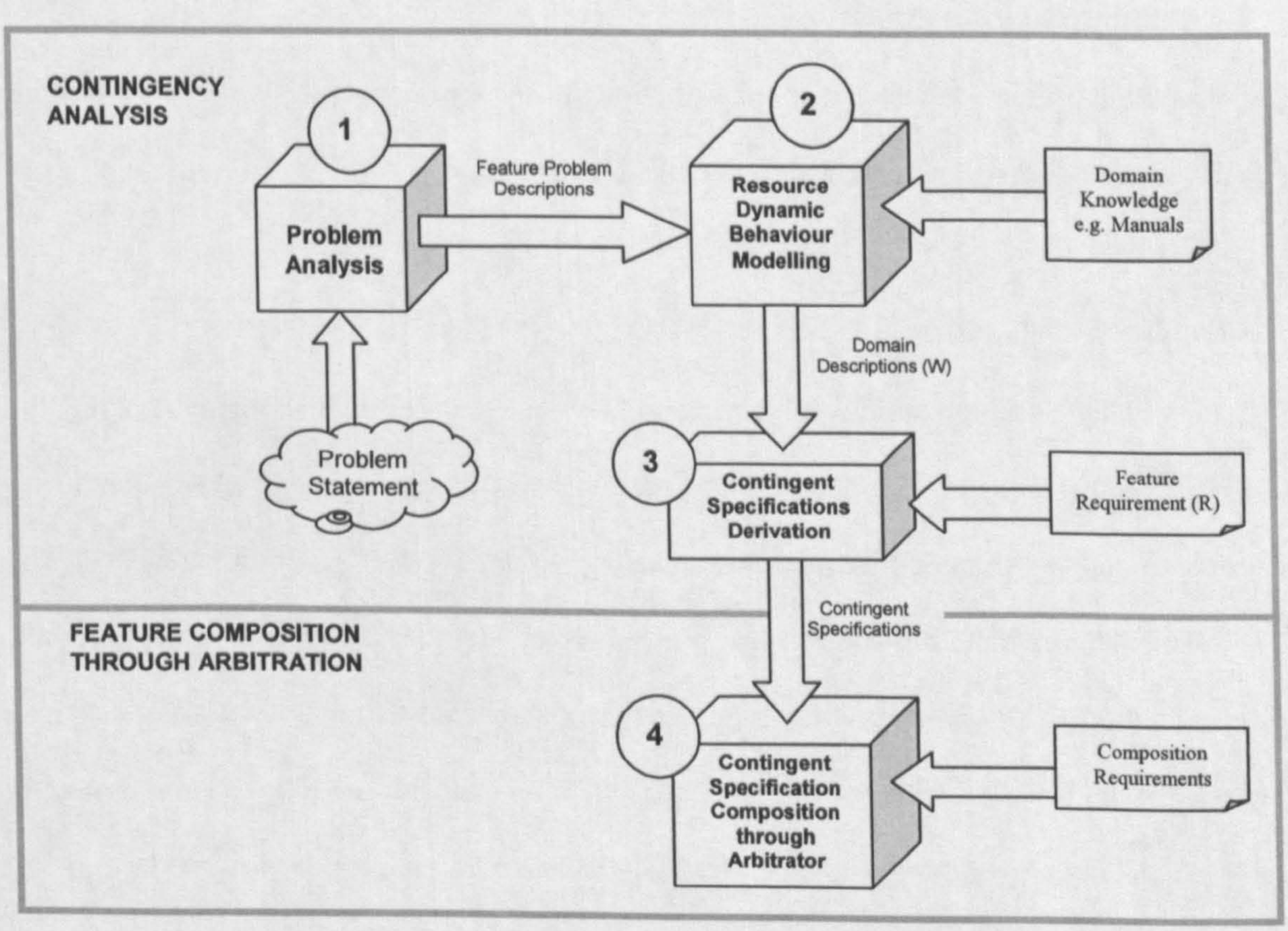


Figure 5.1 Steps Involved in Applying the Proposed Approach

We identify two phases such a development process could entail: (1) *Contingency Analysis*: at design-time each feature is equipped with contingent specifications; and (2) *Feature Composition through Arbitration*: at runtime, the contingent specifications are composed through an arbitrator, called a Composition Controller [Laney *et al.* 2007]. These phases are illustrated in Figure 5.1.

Contingency Analysis consists of 3 steps; namely: *Problem Analysis*, *Resource Dynamic Behaviour Modelling*, and *Contingent Specifications Derivation.* Each contingent specification is executed based on the current state of the shared resource. The second phase entails composing the resulting contingent specifications through an arbitrator. In the following subsections we describe these steps in detail.

5.1 Contingency Analysis

This section presents the first three steps of our approach: problem analysis, resource dynamic behaviour modelling, and contingent specification derivation.

5.1.1 Problem Analysis

Problem analysis involves determining the problem to be addressed by a feature, scoping the context of the problem, and documenting the results of the analysis. Given a requirement, problem analysis enables a requirements analyst to identify and scope the context of the problem. As an illustration consider the problem of providing security for a laptop left on a desk in an office. If we assume that securing the laptop means that only the owner is able to remove it from the desk, then a problem analysis might proceed as follows: *For a thief to remove the laptop from the desk they have to gain entry into the office through the door or windows. Hence one solution to this security problem is to ensure the door and windows are locked and they only open on request from the office owner.*

Initially the context consists of the laptop, office, desk, chair, books, door, and windows. Problem analysis focuses the problem to only the door and windows – reducing the original context. The technical problem that should be solved then becomes: *how do we build a machine that will ensure authorised opening and closing of the door and windows*? The results of problem analysis are expressed as a *problem description*. A problem description shows the relationship between the descriptions of the requirement (R), specification (S), and context (W) [Jackson 2001]. Figures 2.2 and 2.3 are problem descriptions for the burglary capture and burglar deterrence features, respectively.

5.1.2 Resource Dynamic Behaviour Modelling

A *domain description* describes the indicative (i.e. given) behavioural properties of the context [Jackson and Zave 1993]. It maps event occurrences to state changes and thus helps to reason about how successful events result in state changes in the context. We consider an event as successful if it has not been rejected by the arbitrator. Addressing the initialisation problem requires explicit knowledge about the behaviour of the context, states that are possible, and what actions may lead to those states. Domain descriptions are about this information and they are essential in addressing initialisation concerns.

Figure 1.1 is a domain description of the DVD-R. According to this domain description, if the current state of the DVD-R is *Stopped*, the occurrence of a *play* event would result in the state changing to *Playing*. Machines interacting with the DVD-R issue events from the set *{play, record, stop, pause}* to satisfy their requirements. In response to these events, the DVD-R may be in one of the states: *{Playing, Recording, Stopped, Paused_Recording, Paused_Playing}*. In the next section we discuss how our approach applies the concept of contingency planning in preparing feature specifications for satisfying their requirements for different states of a shared resource based on domain descriptions.

5.1.3 Contingent Specification Derivation

In deriving contingent specifications, we use a refinement technique proposed by Laney *et al.* (2007) [Laney *et al.* 2007]. Using this technique, specifications are derived by modelling the behaviour of the context in terms of states and events, and refining requirements over the domain descriptions. We first identify the set of states of the shared domain from its behavioural description. Lets call this set $\beta = \{\beta_1, \beta_2, ..., \beta_k\}$. K is the number of states of the resource. For the DVD-R the set of states are *{Playing, Recording, Stopped, Paused_Recording, Paused_Playing}*.

A requirement describes the desired state of the context to be brought about by a specification. Consequently, we say that a requirement is satisfied when the context is in the state described in the requirement. For example, satisfying the requirement of capturing a burglary requires the DVD-R to be in the *recording* state. In deriving contingent specifications we identify, from β, the state of the shared domain associated with satisfying the requirement. Let us call this state $\alpha \in \beta$. α for the burglary capture requirement is *Recording.*

Given the behavioural descriptions of the shared domain, set of states (β), and the state associated with satisfying the requirement (α), we derive specifications associated with each element in β. Each specification is a sequence of events that should occur if the current state is β_n (where n is in the range 1 to k), to reach α. Table 5.1 shows burglary capture contingent specifications for different initial states of the DVD-R.

Note that the pairs of contingent specifications for the *Paused Recording* and *Paused Playing* states are alternatives. For example, in the case that the initial state is *Paused Recording*, the burglary capture machine either issues a single *record* event or (alternatively) issue a *stop* followed by a *record* event.

Table 5.1. Specifications for Burglary Capture Feature

Initial State	Sequence of Events
Stopped	record
Paused Recording	record
	stop, record
Playing	stop, record
Paused Playing	play, stop, record
	stop, record

5.2 Selecting Contingent Specifications for Composition through an Arbitrator

This section explains step 4 in Figure 5.1. An important question in feature contingency planning is *which of the possible failures due to state inconsistency should be planned for*? This question is especially relevant when a shared resource has a large state space as it may be inefficient and too costly to plan for all possible states.

Our approach assumes that each feature is developed in isolation. The implication of this assumption is that the requirements of other features are not known in advance but become apparent at composition time. Therefore we can now refine our initial question above to: *which of the contingent specifications of a feature should be included when it is composed with other features?* In this section we present selection criteria and a procedure that can be followed in selecting contingent specifications for composition.

Contingency Selection Criteria and Assumptions: Our approach is to plan for all states of the shared resource but include necessary contingent specifications only during composition. The selection criteria targets specifications corresponding to states that will be reached. The first state that will be reached is a *default initial state*. A default initial state is one in which the resource is expected to be when it is initialised. For example, according to Figure 2.2, the default initial state of the DVD-R is *Stopped*. Default initial states are associated with normal

operation. The second set of states included is the set of states reached because of external events issued by other features controlling the resource.

In identifying these states (likely to be reached) we make two assumptions: (1) that changes to states of a resource are due to events issued according to specifications of other features in the composition; and (2) that the specification of each feature in the composition executes to completion. The reasoning behind the first assumption is that state inconsistency results from changes introduced only by other features in the composition.

The second assumption implies that a feature executes successfully, and hence only the *final* state of the resource need consideration in the selection. The final state of the resource with respect to the specification of a feature is the state associated with satisfying its requirement. For example, for the requirement to capture the footage of a burglar using the DVD-R, the final state is *recording*.

Contingency Selection Procedure: Based on the selection criteria and assumptions above, the procedure for selecting the contingent specifications for each feature is outlined below:

Let:

I : be the *default initial state* of the shared resource.

$\delta = \{S_1, S_2, \ldots\ldots, S_k\}$: be the set of contingent specifications for a particular feature.

Where k is the number of contingent specifications for the given feature.

$\Omega = \{\Omega_1, \Omega_2, \ldots\ldots, \Omega_{m-1}\}$: be the set of final states of other features. Where m is the number of features to be composed.

$\Phi = \{\Phi_1, \Phi_2, \ldots, \Phi_n\}$: be the set of contingent specifications, for a given feature, selected for composition.

For each feature:

i. Identify the set of final states of other features in the composition, Ω.

ii. For each element in Ω:

    a. Select a specification from δ whose initial state correspond to that element (Ω_x).

    b. Add the selected specification to Φ.

iii. Select an element in δ whose initial state corresponds to the *default initial state*,

I. Add the selected contingent specification to Φ.

Composing Contingent Specifications: Our approach to composing selected contingent specifications is in two parts: which we refer to as *intra-* and *inter-*composition. *Intra-composition* refers to the composition of contingencies of the same feature, while *inter-composition* refers to the composition of one feature with another through an arbitrator.

Intra-composition assumes that the problem domain (W) can only be in one state at a time, and hence only one of the contingent specifications can be active at a time. Based on this premise, we use the exclusive disjunction (XOR) logical operator to compose contingent specifications of a single feature. The composition of the contingent specifications in relations (4.1) to (4.k) in Chapter 4 is as shown by relation (5.1) below:

$$(S_1 \oplus S_2 \oplus \ldots \oplus S_k),\ W[ß_1, ß_1, \ldots, ß_k] \vdash R \qquad (5.1)$$

The objective to enable the feature to satisfy its requirement (R) by executing *one and only one* of its contingent specifications $\{S_1, S_2, \ldots, S_k\}$ depending on the current state of its context (W). In the next section we present an example that illustrates the steps of the approach presented above.

5.3 Worked Example: Composing Smart Home Features

In this section we illustrate the steps of our proposed approach to the composition of the burglary capture and burglar deterrence features in our running example. In section 5.3.1 we show how the dynamic behaviour of the DVD-R can be described with the Event Calculus. In section 5.3.2 we illustrate the derivation of contingent specifications using a technique that involves refining requirement over domain descriptions. Finally, in section 5.3.3 we apply our contingency selection technique to determine the contingencies that should be included when the two features are composed.

5.3.1 Domain Description of DVD-R

Figure 5.2 shows an EC domain description of the DVD-R. W1 to W4 describe the behaviour of the DVD-R when the current state is *Stopped*. According to W1, the occurrence of a *play* event results in the DVD-R being in the *Playing* state. Similarly, according to W10, the occurrence of a *stop* event changes the state from *PausedPlaying* to *Stopped*. According to W17, the occurrence of a *stop* event results in the DVD-R being in the *Stopped* state regardless of the previous state. This clause (W17) is generic form of the following clauses:

Initiates(stop, Stopped,t) ← HoldsAt(Playing,t) [W17a]

Initiates(stop, Stopped,t) ← HoldsAt(Recording,t) [W17b]

Initiates(stop, Stopped,t) ← HoldsAt(PausedPlaying,t) [W17c]

Initiates(stop, Stopped,t) ← HoldsAt(PausedRecording,t) [W17d]

W17a, W17b, W17c, and W17d correspond to the case in which the current state of the DVD-R is Playing, Recording, PausedPlaying, and PausedRecording, respectively.

initiates(play, Playing, t) ← HoldsAt(Stopped, t)	[W1]
initiates(record, Recording, t) ← HoldsAt(Stopped, t)	[W2]
terminates(play, Stopped, t) ← HoldsAt(Stopped, t)	[W3]
terminates(record, Stopped, t) ← HoldsAt(Stopped, t)	[W4]
initiates(pause, PausedPlaying, t) ← HoldsAt(Playing, t)	[W5]
terminates(pause, Playing, t) ← HoldsAt(Playing, t)	[W6]
terminates(stop, Playing, t) ← HoldsAt(Playing, t)	[W7]
initiates(play, Playing, t) ← HoldsAt(PausedPlaying, t)	[W8]
terminates(play, PausedPlaying, t) ← HoldsAt(PausedPlaying, t)	[W9]
terminates(stop, PausedPlaying, t) ← HoldsAt(PausedPlaying, t)	[W10]
initiates(pause, PausedRecording, t) ← HoldsAt(Recording, t)	[W11]
terminates(pause, Recording,t) ← HoldsAt(Recording, t)	[W12]
terminates(stop, Recording, t) ← HoldsAt(Recording, t)	[W13]
initiates(record, Recording, t) ← HoldsAt(PausedRecording, t)	[W14]
terminates(record, PausedRecording,t) ← HoldsAt(PausedRecording, t)	[W15]
terminates(stop, PausedRecording, t) ← HoldsAt(PausedRecording, t)	[W16]
initiates(stop, Stopped, t)	[W17]

Figure 5.2 Domain Description of DVD-R expressed in Event Calculus

5.3.2 Deriving Smart Home Contingent Specifications

We first formalise the burglary capture and burglar deterrence requirements. Following the refinement method described in *Laney et al.* [Laney *et al.* 2004; Laney *et al.* 2007], we derive contingent specifications of each feature.

Burglary Capture Contingent Specifications: The burglary capture requirement (R_{cap}) can be stated informally as: *When a thief is detected by the burglar sensors, record a video of the burglary*. Using EC, we formalise this requirement as follows:

$$HoldsAt(BurglarDetected, t0) \rightarrow HoldsAt(Recording, t) \wedge (t0 < t) \qquad (R_{cap})$$

The state associated with satisfying R_{cap} is *Recording*. We derive specifications corresponding to each state of the DVD-R by refining the conclusion of R_{cap} over the EC domains description in Figure 5.2. We start with the situation in which the DVD-R is already recording from the surveillance camera. Since *Recording* is already true, we refine the conclusion of R_{cap} using EC1 as follows:

(Refine conclusion by applying EC1)

```
Initially(Recording) ∧ ¬Clipped(t0, Recording, t)
```

(Apply EC3 to the second sub-clause)

```
Initially(Recording) ∧ ¬∃e1,t1 · happens(e1,t1) ∧
terminates(e1, Recording, t1) ∧ (t0<t1<t)
```

(Unify the terminate sub-clause with W12 and W13)

```
Initially(Recording) ∧ ¬∃ t1 · happens(pause,t1) ∧
terminates(pause, Recording, t1) ∧ ¬∃ t1 · happens(stop,t1) ∧
terminates(stop, Recording, t1) ∧ (t0<t1<t)
```

(Remove the terminate clauses since they are axioms)

```
Initially(Recording) ∧ ¬∃ t1 · happens(pause,t1) ∧
¬∃ t1 · happens(stop,t1) ∧ (t0<t1<t)
```

(Replace the second and third clauses with prohibit clauses)

```
Initially(Recording)∧prohibit(pause,t0,t) ∧prohibit(stop,t0,t)
```

From the derivation above, the burglary capture specification corresponding to the *Recording* state is:

```
HoldsAt(BurglarDetected, t0) →
```

```
Initially(Recording) Λ prohibit(pause,t0,t) Λ
prohibit(stop,0,t) Λ (t0 < t)
```

Since the *Initially(Recording)* clause cannot be enforced by the specifications and it is condition that should be true before the *prohibit(...)* actions can be executed, we re-write the specification with this clause as a pre-condition.

HoldsAt(BurglarDetected, t0) Λ HoldsAt(Recording, t0) →

prohibit(pause,0,t) Λ prohibit(stop,0,t) Λ (t0 < t) (SS1)

We derive the burglary capture specifications corresponding to the rest of the states by applying EC2 to the conclusion of R_{cap}. We follow similar steps of refinement as shown above. The resulting contingent specifications for the burglary capture feature are shown in Table 5.2.

Table 5.2. Specifications for the Burglary Capture Feature

	Current State	Specification
SS1	Recording	HoldsAt(BurglarDetected, t0) Λ HoldsAt(Recording, t0) → prohibit(pause,0,t) Λ prohibit(stop,0,t) Λ (t0<t)
SS2	Stopped	HoldsAt(BurglarDetected, t0) Λ HoldsAt(Stopped, t0) → happens(record,t1) Λ prohibit(pause,t2,t) Λ prohibit(stop,t2,t) Λ (t0< t1<t2<t)
SS3a	Playing	HoldsAt(BurglarDetected, t0) Λ HoldsAt(Playing, t0) → happens(stop,t1) Λ happens(record,t2) Λ prohibit(play,t2,t3) Λ prohibit(stop,t3,t)Λ prohibit(pause,t3,t) Λ (t0<t1<t2<t3<t)
SS3b	Playing	HoldsAt(BurglarDetected, t0) Λ HoldsAt(Playing, t0) → happens(pause,t1) Λ prohibit(play,t2,t4) Λ happens(stop,t2) Λ happens(record,t3) Λ prohibit(stop,t4,t) Λ prohibit(pause,t4,t) Λ (t0<t1<t2<t3<t4<t)
SS4a	Paused-Playing	HoldsAt(BurglarDetected, t0) Λ HoldsAt(PausedPlaying, t0) → happens(stop,t1) Λ prohibit(play,t2,t3) Λ happens(record,t2) Λ prohibit(stop,t3,t) Λ prohibit(pause,t3,t) Λ (t0<t1<t2<t3<t)
SS4b	Paused-Playing	HoldsAt(BurglarDetected, t0) Λ HoldsAt(PausedPlaying, t0) → happens(play,t1) Λ prohibit(pause,t2,t3) Λ happens(stop,t2) Λ prohibit(play,t3,t4) Λ happens(record,t3) Λ prohibit(stop,t4,t) Λ prohibit(pause,t4,t) Λ (t0<t1<t2<t3<t4<t)
SS5a	Paused-Recording	HoldsAt(BurglarDetected, t0) Λ HoldsAt(PausedRecording, t0)→ happens(record,t1) Λ prohibit(stop,t2,t) Λ prohibit(pause,t2,t) Λ (t0<t1<t2<t)
SS5b	Paused-Recording	HoldsAt(BurglarDetected, t0) Λ HoldsAt(PausedRecording, t0)→ happens(stop,t1) Λ prohibit(play,t2,t3) Λ happens(record,t2,t) Λ prohibit(stop,t3,t) Λ prohibit(pause,t3,t) Λ (t0<t1<t2<t3<t)

Similar to Table 5.1, the pairs of contingent specifications SS3a - SS3b, SS4a - SS4b, and SS5a - SS5b are alternatives. Each pair corresponds to the same initial state. For example, both SS5a and SS5b are contingencies for state *PausedRecording*. According to SS5a if a burglar is detected while the DVD-R is in the PausedRecording state, the burglary capture machine should issue a *record* event and thereafter the composition controller should prohibit *stop* and *pause* events as they will violate the requirement to record a burglary.

Similarly, according to SS3b, if a burglar is detected and the DVD-R is in the playing state, then the burglary capture machine should issue a *pause* event. This takes the DVD-R to the *PausedPlaying* state. Once in the *PausedPlaying* state the composition controller should prohibit *play* events as this will reverse the previous action. The burglary capture machine that issues a record event, to start recording the burglary and the composition controller is instructed to reject *stop* and *pause* events as their occurrence will violate the recording requirement.

Burglary Deterrence Contingent Specifications: We state the deterrence requirement (R_{det}), informally, as follows: *If the house owner is away, play a movie.* Again using EC the requirement is stated formally as:

$$\text{HoldsAt(AwayFromHome, t)} \rightarrow \text{HoldsAt(Playing, t)} \qquad (R_{det})$$

Following the same refinement method used for deriving burglary capture specifications, above, we derived the specifications for the burglar deterrence feature shown in Table 5.3. The pairs ES3a - ES3b, ES4a - ES4b, and ES5a - ES5b are alternative contingent specifications.

At run-time, the contingent specifications are selected based on the current state of the DVD-R. As an illustration of the selection of the correct specifications, consider a hypothetical

scenario in which the burglar deterrence feature is in control of the DVD-R. Under the control of this feature the DVD-R is in the *Playing* state. Assume that burglary capture has a higher priority than burglar deterrence. If a thief breaks-in, deterrence is pre-empted by burglar capture to record the burglary. The burglary capture machine selects either SS3a or SS3b.

Table 5.3. Specifications for Burglar Deterrence Feature

	Current State	Specification
ES1	Playing	HoldsAt(AwayFromHome, t0) Λ HoldsAt(Playing, t0) → prohibit(pause,0,t) Λ prohibit(stop,0,t) Λ (t0<t)
ES2	Stopped	HoldsAt(AwayFromHome, t0) Λ HoldsAt(Stopped, t0) → happens(play,t1) Λ prohibit(pause,t2,t) Λ Prohibit(stop,t2,t) Λ (t0<t1<t2<t)
ES3a	Recording	HoldsAt(AwayFromHome, t0) Λ HoldsAt(Recording, t0) → happens(stop,t1) Λ prohibit(record,t2,t3) Λ happens(play,t2) Λ prohibit(pause,t3,t) Λ prohibit(stop,t3,t) Λ (t0<t1<t2<t3<t)
ES3b	Recording	HoldsAt(AwayFromHome, t0) Λ HoldsAt(Recording, t0) → happens(pause,t1) Λ prohibit(record,t2,t4) Λ happens(stop,t2) Λ happens(play,t3) Λ prohibit(pause,t4,t) Λ prohibit(stop,t4,t) Λ (t0<t1<t2<t3<t4t)
ES4a	Paused-Recording	HoldsAt(AwayFromHome, t0) Λ HoldsAt(PausedRecording, t0) → happens(stop,t1) Λ prohibit(record,t2,t3) Λ happens(play,t2) Λ prohibit(pause,t3,t) Λ prohibit(stop,t3,t) Λ (t0<t1<t2<t3<t)
ES4b	Paused-Recording	HoldsAt(AwayFromHome, t0) Λ HoldsAt(PausedRecording, t0) → happens(record,t1) Λ prohibit(pause,t2,t3) Λ happens(stop,t2) Λ prohibit(record,t3,t4) Λ happens(play,t3) Λ prohibit(pause,t4,t) Λ prohibit(stop,t4,t) Λ (t0<t1<t2<t3<t4t)
ES5a	Paused-Playing	HoldsAt(AwayFromHome, t0) Λ HoldsAt(PausedPlaying, t0) → happens(stop,t1) Λ prohibit(record,t2,t3) Λ happens(play,t2) Λ prohibit(pause,t4,t) Λ prohibit(stop,t4,t) Λ (t0<t1<t2<t3<t4t)
ES5b	Paused-Playing	HoldsAt(AwayFromHome, t0) Λ HoldsAt(PausedPlaying, t0) → happens(play,t1) Λ prohibit(pause,t4,t) Λ prohibit(stop,t4,t) Λ (t0<t1<t2<t3<t4t)

When the selected burglary capture specification has finished executing it leaves the DVD-R in the *Recording* state. While suspended, the burglar deterrence machine monitors state changes in the DVD-R. When it resumes execution it selects either specification ES3a or ES3b (which corresponds to the Recording state). Although the state of the DVD-R has been changed by the burglary capture feature, the burglar deterrence feature still satisfies its

requirement by selecting the correct specification. In chapter 7 we present simulation results showing arbitration with and without contingencies.

5.3.3 Selecting Smart Home Contingencies for Composition

In selecting contingent specifications for composition we follow the procedure outlined in section 5.2. We start by selecting contingencies for the burglary capture feature:

- The initial state of the DVD-R *I*, is *Stopped.*
- The set of contingent specification for the burglary capture feature δ_{sec} = *{SS1, SS2, SS3a, SS3b, SS4a, SS4b, SS5a, SS5b}*
- The set of states associated with satisfying the burglar deterrence requirement Ω = *{Playing}.*
- Based on the three sets: *I*, δ_{sec}, *and* Ω; the set of selected contingent specifications for burglary capture, Φ_{sec} = *{SS2, SS3a, SS3b}.*

Note that SS2 corresponds to the initial state, *Stopped.* Meanwhile, SS3a and SS3b both correspond to the *Playing* state. By following the same method above the set of selected contingent specifications for the burglar deterrence feature Φ_{ent} = *{ES2, ES3a, ES3b}.*

5.4 Chapter Summary

This chapter has illustrated how our proposed conceptual approach to the initialisation problem could be applied in practice by showing how an existing arbitration approach, the Composition Controller [Laney *et al.* 2007], can be extended with contingencies. We have presented a development process that could be used in developing a feature-based application that makes use of the concepts of arbitration and contingency planning to resolve runtime feature interactions. We have identified two main steps such a development process could entail, namely: (1) contingency analysis which involves building contingencies into

specifications and (2) composing the contingent specifications through arbitration. We have illustrated our approach to the composition of smart home features. In the next chapter we discuss tool support for the derivation of contingent specifications.

Chapter 6. Tool Support for Deriving Contingent Specifications

In Chapter 5 we showed how to derive contingent specifications and compose them through a Composition Controller. The derivation of contingency specifications can be time-consuming and error prone, if done manually, hence the need for automation. In this chapter we present a tool, called *Contingency Specification Generator (CSG)* that automates the contingency specification derivation process. CSG is based on Graph Theory [Gould 1988] and abduction [Russo *et al.* 2002]. Abduction is a reasoning technique for determining what events might lead from an initial state to a final state [Mueller 2006a]. We express a requirement in terms of the set of fluents that should hold in the final state. The initial state is expressed in terms of the fluents that currently hold. Figure 6.1 shows, at a high-level, how the tool generates contingent specifications.

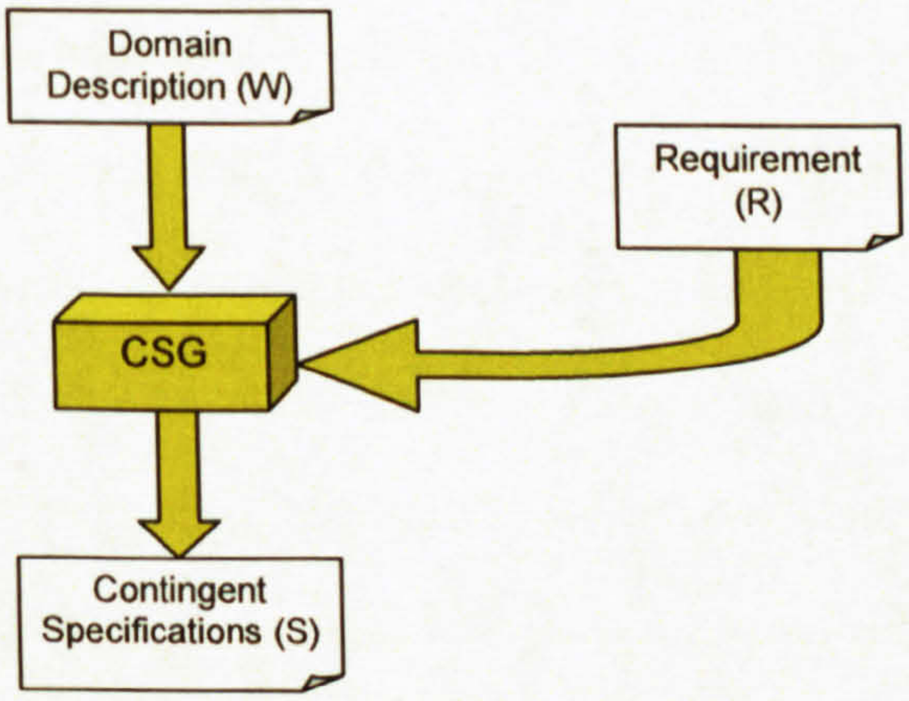


Figure 6.1 High-level Architecture of the Contingent Specification Generator (CSG) tool

CSG takes as input EC domain descriptions (W) and the requirement (R). The domain descriptions are converted into directed graphs. Abduction is then applied on the directed graph to refine the requirement into contingent specifications (S). The specifications are expressed in EC and stated in terms of: (a) the sequence of events that should occur in order

to satisfy a requirement given the current state as an initial state; and (b) events that should be prohibited in order to sustain the satisfaction of the requirement. We discuss, in detail, the derivation of contingent specifications through abductive reasoning in section 6.1 and compare our tool to existing tools in section 6.2. Finally, we present a summary of the chapter in section 6.3.

6.1 Deriving Specifications through Abductive Reasoning on Directed Graphs

We explain the tool in detail in the following sections. Section 6.1.1 discusses the conversion of an EC description into a directed graph. Section 6.1.2 explains how abduction is applied on the directed graphs to identify paths. In section 6.1.3 the paths are re-expressed as EC specifications.

6.1.1. Converting Event Calculus Descriptions to Directed Graphs

We chose to translate the EC description into directed graphs for two reasons: (1) the availability of mature algorithms for analysing directed graphs; (2) the ease with which such algorithms can be implemented. A directed graph consists of vertexes connected by edges. In translating an EC description to a directed graph, we associate each vertex with a set of fluents that should hold. We use the edges to represent transitions between vertexes and hence changes in fluent values. Each edge is labelled with the event that should occur for a transition to occur. Therefore, a transition is described in terms of three entities: *Transition (Current State, Event, Next State)*. *Current State* is the set of fluents that hold in the current vertex. *Next State* is the set of fluents that would hold as a result of the transition. *Event* is the action that should happen to trigger the transition.

A directed graph representation of the EC descriptions in Figure 5.2 is shown in Figure 6.2. We have used Figure 6.2 here as a visual aid for explaining the concepts of translation from EC descriptions to directed graphs. In the CSG tool such a graphical representation is not

necessary as the tool represents the directed graph as a collection of transitions as shown in Figure 6.3.

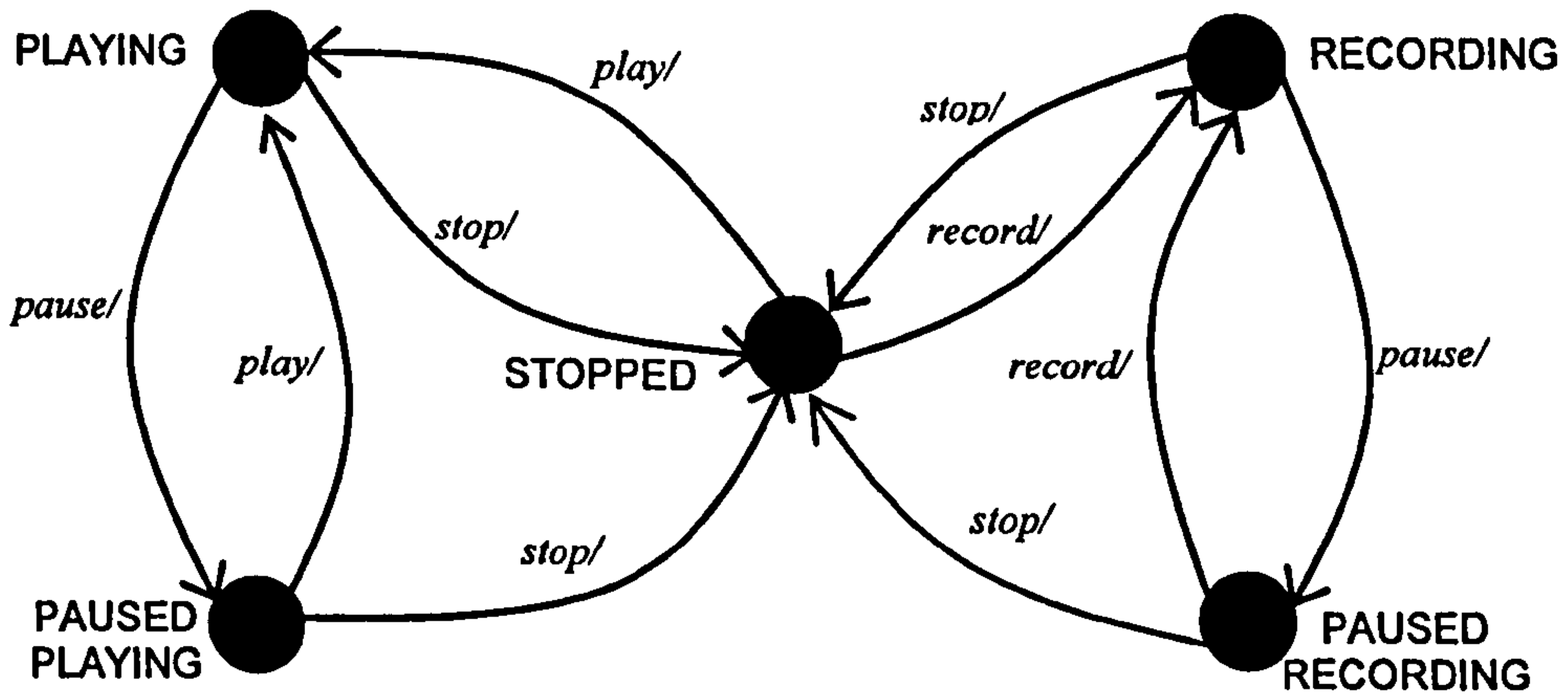


Figure 6.2 Directed Graph Description of DVD-R Behaviour

Transition(STOPPED, stop, STOPPED)	(WDG_1)
Transition(STOPPED, play, PLAYING)	(WDG_2)
Transition(STOPPED, record, RECORDING)	(WDG_3)
Transition(STOPPED, pause, STOPPED)	(WDG_4)
Transition(PLAYING, stop, STOPPED)	(WDG_5)
Transition(PLAYING, play, PLAYING)	(WDG_6)
Transition(PLAYING, record, PLAYING)	(WDG_7)
Transition(PLAYING, pause, PAUSED_PLAYING)	(WDG_8)
Transition(RECORDING, stop, STOPPED)	(WDG_9)
Transition(RECORDING, play, RECORDING)	(WDG_{10})
Transition(RECORDING, record, RECORDING)	(WDG_{11})
Transition(RECORDING, pause, PAUSED_RECORDING)	(WDG_{12})
Transition(PAUSED_PLAYING, stop, STOPPED)	(WDG_{13})
Transition(PAUSED_PLAYING, play, PLAYING)	(WDG_{14})
Transition(PAUSED_PLAYING, record, PAUSED_PLAYING)	(WDG_{15})
Transition(PAUSED_PLAYING, pause, PAUSED_PLAYING)	(WDG_{16})
Transition(PAUSED_RECORDING, stop, STOPPED)	(WDG_{17})
Transition(PAUSED_RECORDING, play, PAUSED_RECORDING)	(WDG_{18})
Transition(PAUSED_RECORDING, record, RECORDING)	(WDG_{19})
Transition(PAUSED_RECORDING, pause, PAUSED_RECORDING)	(WDG_{20})

Figure 6.3 Directed Graph description of DVD-R represented as Transitions

Since we are translating Event Calculus domain descriptions into finite state machine behavioural descriptions, our work focuses on the form of initiates(...) and terminates(...) EC clauses shown in Figure 5.2. In the following we discuss how these forms of EC clauses are translated into state machine descriptions.

Each *initiate*(e,F_1,t) ← *HoldsAt*(F_2,t) clause states that the occurrence of event *e* initiates fluent F_1 provided fluent F_2 currently holds. Based on the relationship between fluents and vertexes described above, this EC clause is translated to *Transition*(F_2,e,F_1) in a directed graph. For example the EC clause *Initiates (play, Playing, t)* ← *HoldsAt (Stopped, t), W1* in Figure 5.2, is expressed as *Transition(Stopped, play, Playing), WDG2 in Figure 6.3*. The mapping of all the initiate(...) clauses in Figure 5.2 to the transitions in Figure 6.3 is as shown in Table 6.1.

Table 6.1. Mapping of Initiates(...) Clauses to Transitions

Event Calculus Initiate Clause	State Transition Mapping
W1	WDG2
W2	WDG3
W5	WDG8
W8	WDG14
W11	WDG12
W14	WDG19
W17a	WDG5
W17b	WDG9
W17c	WDG13
W17d	WDG17

In the translation from an Event Calculus to a finite state machine description terminate(...) clauses do not need to be included explicitly since they are implied in initiate(...) clauses. This is because a transition in a finite state machine directly describes initiating actions and, additionally, indirectly implies terminating actions. More precisely this could be described as

follows: *In a state machine description, the transition from some state X to another state Y due to the occurrence of an event e initiates fluents in state Y, while at the same time terminating fluents associated with state X.*

Consequently, while it is necessary to explicitly include both initiates(...) and terminates(..) clauses in Event Calculus descriptions, it is sufficient to translate only initiates(...) clauses for equivalent finite state machine behavioural description. This is because transitions describe both fluent initiating and terminating actions. For example, in Figure 5.2, clause W3 is implied in clause W1 and hence transition WDG2 in Figure 6.3 describes both clauses. The mapping of all the terminates(...) clauses in Figure 5.2 to the transitions in Figure 6.3 is as shown in Table 6.2 below:

Table 6.2. Mapping of Terminates(...) Clauses to Transitions

Event Calculus Initiate Clause	State Transition Mapping
W3	WDG2
W4	WDG3
W6	WDG8
W7	WDG5
W9	WDG14
W10	WDG13
W12	WDG12
W13	WDG9
W15	WDG19
W16	WDG17

Figures 1.1 and 6.2 are incomplete descriptions of the DVD-R behaviour because they do not show how the occurrences of events that have no outgoing transitions from a given state are handled. For example they do not answer the question: if the current state is *Playing* and a *play* event occurs, what should be the next state? This is handled by transition WDG6 in Figure 6.3. According to WDG6 if a play event occurs while the current state is Playing, there

is no change of state. Other similar error handling transitions are WDG1, WDG4, WDG7, WDG10, WDG11, WDG15, WDG16, WDG18, and WDG20. These transitions have been deliberately omitted from Figures 1.1 and 6.2 to keep the graphical representations simple and improve readability.

In the next section we show how our tool uses a directed graph analysis algorithm to implement abduction.

6.1.2 Using Abduction to Identify Paths through a Graph

We use Dijkstra's Distance Algorithm [Gould 1988] to traverse the directed graph and find all paths from a source to a target vertex. The source vertex (node) is associated with the current state, while the target vertex is associated with next state. Each path consists of a sequence of transitions (edges). For example, in Figure 6.2, if initially *PausedPlaying* holds and it is desired that fluent *Recording* should be true, there are two paths, p1 and p2 shown below:

P1: Transition(PausedPlaying, stop, Stopped), Transition(Stopped, record, Recording).

P2: Transition(PausedPlaying, play, Playing), Transition(Playing, stop, Stopped),
Transition(Stopped, record, Recording).

An exhaustive list of paths is obtained by selecting each of the different fluents as sources to the target until all fluents have been explored. This ensures that the specification can cause a desired fluent to hold regardless of the current state of the shared domain.

6.1.3 Expressing Paths as Specifications

Finally, each path is converted into a specification by extracting the events in each transition. The specifications are then re-expressed in the Event Calculus. We re-express the specifications in EC for convenience and to take advantage of the rich vocabulary of the EC. The re-expression is based on the idea that each transition in a directed graph can be expressed as three EC clauses:

- happens(event, t)
- terminates(event, CurrentState, t) ← holdsAt(CurrentState, t)
- initiates(event, NextState, t) ← holdsAt(CurrentState, t)

For example *Transition (Stopped, record, Recording)* in path P1 (section 6.1.2) can be re-expressed as:

- happens(record, t),
- terminates(record, Stopped, t) ← holdsAt(Stopped, t) $[W_4]$
- initiates(record, Recording, t) ← holdsAt(Stopped, t) $[W_2]$.

Informally, this can be stated as: if the DVD-R is currently *Stopped*, the occurrence of a *record* event at time *t* results in fluent *Stopped* being *false* and fluent *Recording* being *true*. In a similar manner to the refinement method described in Laney *et al.*[Laney *et al.* 2007], we do not include *terminates(...)* and *initiates(...)* clauses in the specification since they are part of the axioms. The truth values of these clauses are taken for granted since they are part of the (indicative) domain descriptions. They serve as a starting point from which we logically derive specifications. We include the *happens(...)* clauses in the specification since they state the actions that should happen for the requirement to be satisfied. The right hand sides in W_4 and W_2 are retained as pre-conditions to the occurrence of the event in the happens(...) clause.

Deviations from a given path may lead to the violation of a requirement. Such deviations are possible if the current vertex in a path has more than one outgoing edge. To avoid these deviations we use the *prohibit(...)* [Laney *et al.* 2007] clause to instruct the composition controller to reject events that lead to different transitions which are not part of the current path. This yields the contingent specifications associated with each possible state of the problem domain. Figure 6.4 shows screen shot of a partial view of a text file representing contingent specifications generated by the CSG tool for the Burglary Capture feature.

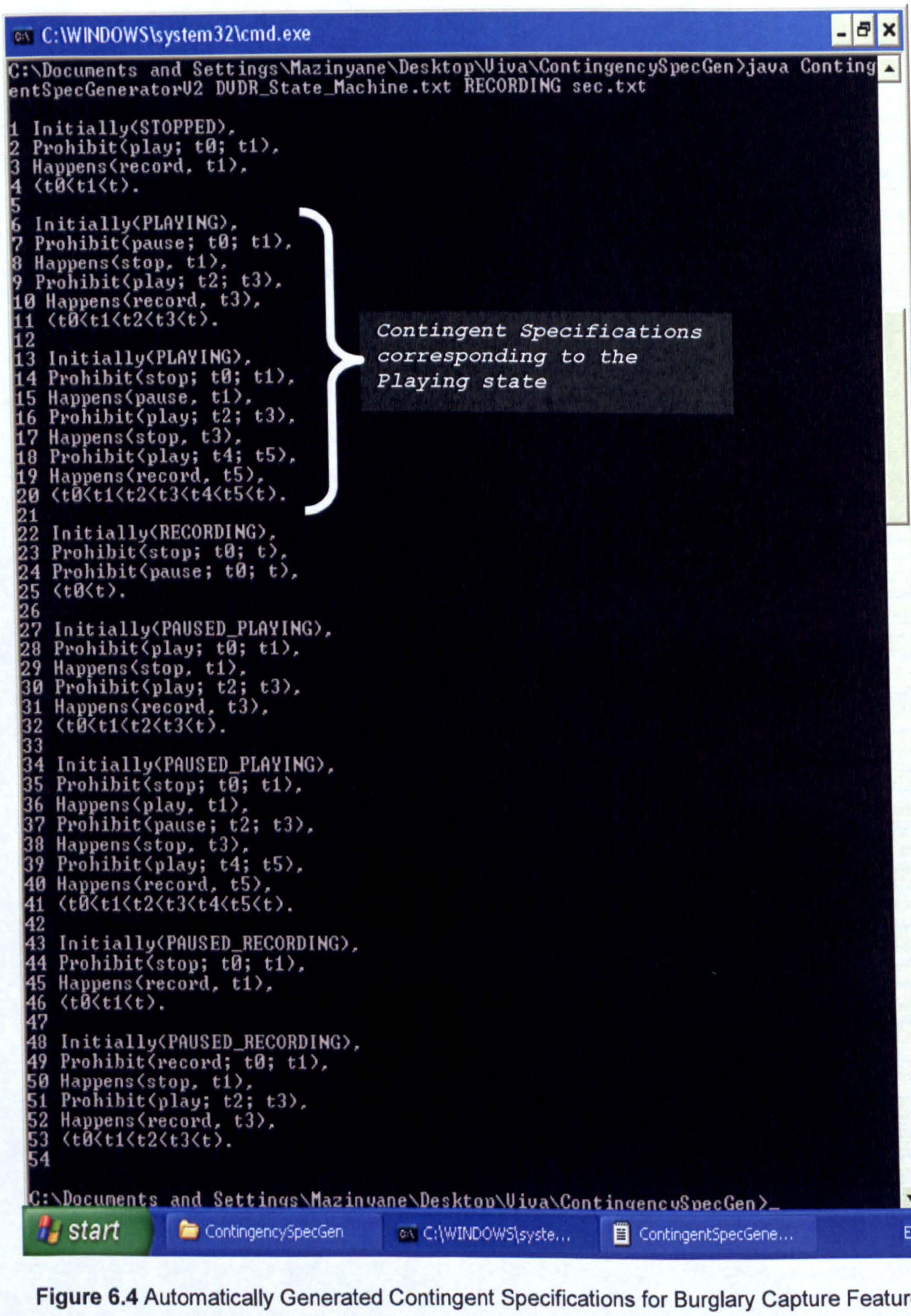


Figure 6.4 Automatically Generated Contingent Specifications for Burglary Capture Feature

Each paragraph represents a contingent specification. The *initially(....)* clauses in each contingent specification represents the state that should hold for the corresponding specification to be executed. For example lines 6 – 20 represent two contingent specifications

that would be executed if the current state of the DVD-R is *Playing* to satisfy the burglary capture requirement. The contents of the first contingent specification are shown between lines 6 – 11 and they correspond to SS3a in table 5.2. We will use this specification as an exemplar to illustrate the contents of Figure 6.4. The specification runs from time t0 to t3.

Line 7 states that at time t0 the burglary capture machine issues instruction to the Composition Controller to prohibit *pause* events between t0 and t1. At t1 the burglary capture machine issues a *stop* event (line 8) and at t2 an instruction is issued for the Composition Controller to reject play events between times t2 and t3 (line 9). In line 10, the burglary capture machine issues a *record* event instructing the DVD-R to start recording. Line 11 shows the order from time t0 to t3.

6.3. Comparison to the Event Calculus Planner

Our tool is inspired by the Event Calculus Planner (ECP) [Shanahan 1999]. Given a domain behavioural description expressed in the EC and the requirement to be satisfied expressed as fluents that should hold, the planner automatically generate the sequence of events (plans) that satisfy the requirement. Our tool works in a similar manner. The main difference between the two is that the CSG is specific for generating contingent specifications while the ECP is a generic AI planner.

In generating specifications we do not only consider the behaviour of the context, but we are also explicit about the initial state of the context. This makes it possible to explore, exhaustively, the different states of the context and derive corresponding contingencies. The same effect can be achieved by integrating our tool with the ECP such that ECP acts as an engine for generating single contingent specifications based on parameters supplied by the CSG.

The differences between state machines and the Event Calculus descriptions are not significant in our approach to deriving contingencies. This is because the derived specifications are the same regardless of the language (whether state machine or Event Calculus) used initially for describing the behaviour of a problem domain.

6.4 Chapter Summary

We have presented automated tool support for deriving contingent specifications. The contingent specifications are part of our approach to composing and resolving feature interactions at runtime. The tool is based on graph theory and derives specifications by performing abductive reasoning on a dynamic description of a resource. The next chapter presents an evaluation of our approach to the initialisation problem through its application to both our running example and a practical problem.

Chapter 7. Evaluation

Our main claim, in this thesis, is that combining arbitration with contingency planning ensures that, whenever a conflict occurs, the requirement of a feature that is eventually granted access to a shared resource will be satisfied. In section 7.1 we test the validity of this claim by simulating the contingent specifications of the smart home derived in Chapter 5. We have also validated the industrial applicability of our approach by applying it to the resolution of conflicts between features of a computer-based automotive application. We present the results of our validation in section 7.2 and a summary of this chapter in Section 7.3.

7.1 Evaluation on Smart Home Features

Using CSG we derived the contingent specifications for the burglary capture and burglar deterrence features. The specifications and Composition Controller were then implemented in Java. The dynamic behaviour of the DVD-R was modelled using ECharts [Bond]. ECharts is a state-machine based programming language for event driven systems. Appendix 1 shows the coding of the DVD-R domain descriptions in ECharts. In this section we present simulation results when the contingent specifications were composed through the Composition Controller (section 7.1.1) and a discussion of the results (section 7.1.2).

7.1.1 Smart Home Specification Simulation Results

Table 7.1 and 7.2 show simulation results when the burglary capture and burglar deterrence features were composed *sequentially,* with burglary capture having a higher priority than burglar deterrence. Recall that: the burglary capture requirement (R_{cap}) is to record a burglary, while the deterrence (R_{det}) is to play a movie. S_{cap} and S_{det} are the specifications of the capture and deterrence features, respectively. W_s indicates the current state of the DVD-R. The post-

condition of the deterrence requirement is that the DVD-R is in the *Playing* state, while the post-condition of capture requirement is that the DVD-R is in the *Recording* state.

When composed sequentially these features exhibit a bypass feature interaction. To illustrate this, assume that for both features the pre-states are that the DVD-R is in the *Stopped* state. Consider a scenario in which burglar deterrence is executed at 6 pm and the movie last for 3 HRS. If a burglary happens at 8 pm, the burglary capture feature will be triggered through the burglar sensors but will fail to execute. This is because at this time the DVD-R will be in the Playing state and this is inconsistent with burglary capture's pre-state. As a result the capture requirement is not satisfied and the deterrence feature is said to have bypassed the capture feature.

Table 7.1 shows the simulation results when the Composition Controller is used with feature specifications that have no contingencies. The first row shows the time range in terms of timepoints over which the simulation was executed. There are four timepoints indicated by the columns labelled 0 to 3. At each timepoint the events issued by each feature, the status of each feature's requirement (satisfied or unsatisfied) and the current state of the DVD-R are recorded. This enabled us to monitor the satisfaction of the requirements of features in response to the occurrence of events issued according to the feature specifications.

Table 7.1. Composition Controller without Contingencies

Time:	0	1	2	3
S_{cap}	Prohibit(play;t0;t1)	Record	-	-
R_{cap}	Unsatisfied	Satisfied	Satisfied	Satisfied
S_{det}	-	-	Prohibit(record;t0;t1)	Play
R_{det}	-	-	Unsatisfied	Unsatisfied
W_s	STOPPED	RECORDING	RECORDING	RECORDING

The row labelled S_{cap} shows the sequence of events issued by the burglary capture feature, while the row labelled R_{cap} shows whether the burglary capture requirement is satisfied or unsatisfied at a given timepoint. Similarly, the row labelled S_{det} shows the sequence of events issued by the deterrence feature while the row labelled R_{det} shows whether the deterrence requirement is satisfied or unsatisfied at a given timepoint. We have used the dash ("-") to indicate that at the given timepoint no event has been issued by a feature and hence its requirement is not satisfied. This is the case with the rows labelled S_{det} and R_{det} at timepoints 0 and 1.

The specifications of both features had *Stopped* as the pre-state. Initially the DVD-R is in the *Stopped* state. When a non-deterministic conflict occurs on the DVD-R, the burglary capture feature is granted access because of its higher priority. Since the current state of the DVD-R matches the preconditions of the capture specification, the execution of S_{cap} satisfies R_{cap} by leaving the DVD-R in the *Recording* state at timepoint t=1. However, when the deterrence feature starts executing at t=2, its requirement is not satisfied because the current state is inconsistent with its pre-state. Note that this also implies that if the preconditions of the higher priority feature were not met then both requirements would not be satisfied – rendering the effort of arbitration futile.

Table 7.2. Composition Controller with Contingencies

Time:	0	1	2	3	4	5
S_{cap}	Prohibit(play;t0;t1)	Record	-	-	-	-
R_{cap}	Unsatisfied	Satisfied	Satisfied	-	-	-
S_{det}	-	-	Prohibit(pause;t2;t3)	stop	Prohibit(record;t4;t5)	Play
R_{det}	-	-	Unsatisfied	Unsatisfied	Unsatisfied	Satisfied
W_s	STOPPED	RECORDING	RECORDING	STOPPED	STOPPED	PLAYING

Table 7.2 shows the results of simulation when the Composition Controller is composing contingent specifications. Similar to table 7.1, table 7.2 shows timepoints as column headers

(0-5), and events issued according to feature specifications, status on satisfaction of requirements, and states on the DVD-R as row labels.

Contrary to table 7.1, according to table 7.2, the deterrence requirement is also satisfied (at time t=5, row 5) because a contingent specification corresponding to the *Recording* state of the DVD-R was selected at timepoint t=2. Hence, despite the use of arbitration to resolve non-determinism both requirements are eventually satisfied. This substantiates our claim that combining arbitration with contingency planning guarantees that the requirements of conflicting features are eventually satisfied (where possible).

7.1.2 Discussion of Smart Home Evaluation Results

We have illustrated how contingency planning complements arbitration by ensuring that the requirement of a feature that is granted access to a shared resource is eventually satisfied. In this section we discuss the limitations of our approach by examining the assumptions we have made and the implications of having those assumptions violated. This includes assumptions on using a model to track state changes, contingency selection criteria, selective contingency derivation, and location of contingencies.

Using Model of a Resource to Track State Changes: The model of a resource is central to the ability of a feature to determine a resource state. Updating models with successful events does not guarantee that the state in the model accurately represents the actual (physical) state of the world. The assumption is that if an event was not prohibited by the arbitration process then it has caused the necessary effect in the real world. This assumption may not always be true because of the possibility that a successful event that has been used to update the model may not reach the real world.

For example, the burglary capture machine and the DVD-R share events through interface c in Figure 2.2. This interface could be realised through a physical data cable. If the burglary capture machine issues a *record* event and this event is not prohibited by the arbitrator, the assumption is that it has caused the corresponding effect on the DVD-R. However, if the cable is broken, this assumption does not hold and the model will not be consistent with the actual state of the DVD-R! Addressing these concerns may need a monitoring [Fickas and Feather 1995] mechanism on the shared domain. We have ignored these concerns by assuming that every successful event results in the corresponding state changes in the resource.

Validity of Assumptions on Contingency Selection Criteria: The contingency selection criteria described in chapter 5 assumes that (inconsistent) state changes in a shared resource result from events issued according to specifications of features in the composition. This assumption ignores state changes that may result from actions issued by external agents. An example of an external agent is a person pressing the *play* button on the DVD-R.

The cost of ignoring external events is that it is difficult to deal with state changes that do not originate from within the features currently in the composition or that have not resulted from an event which is part of their (features) collective set of events. If one contingent specification has *{play, stop}* and another has *{record, stop}*, their collective set of events is *{play, stop, record}*. The occurrence of these events would result in the DVD-R being in the *Playing*, *Stopped*, and *Recording* states, respectively.

The contingent specifications included in the composition would correspond to these three states. The occurrence of an event outside this set would result in the DVD-R being in a state with no corresponding contingent specification. For example, a *pause* event would leave the DVD-R either in the *PausedPlaying* or *PausedRecording* state. Since none of the contingent specifications has these two states as their initial states, it may not be possible to recover from

these states. In addressing these issues, the work on obstacle analysis [van Lamsweerde and Willemet 1998] may be a starting point.

Selective Contingency Derivation and Intermediary States: Our approach is to plan for all contingencies and eliminate those not necessary when features are composed. If there is high cost associated with planning for each contingency, eliminating the contingencies during composition may already be too late to avoid such a cost. Thus another option is to be selective from the onset on which states should be provided with contingencies. However, this option is not compatible with one of the main assumptions of our approach. As discussed in section 5.2, we assume that each feature is developed in isolation. Thus it is not possible to decide in advance which contingencies to include in the planning as the requirements for other features can only be known during composition.

Our contingency selection technique also assumes that features execute to completion. This assumption does not always hold. One situation where it may be invalid is when a higher priority feature pre-empts a lower priority feature such that the resource is left in an *intermediary* state instead of a final state. Recall that in the case where a feature specification executes successfully only the *final* state of the resource needs consideration in the selection.

Meanwhile in the case where a specification may not complete execution, intermediary states of the resource are considered. An intermediary state is reached if a specification does not execute to completion; that is, it does not reach the final state. This is likely to happen if a feature is suspended or its execution aborts (such is the case with pre-emption). For example, assuming the current state of the DVD-R is *Playing*, to satisfy the requirement of capturing a burglary the DVD-R changes from *Playing*, to *Stopped*, and then to *Recording*. *Playing* is the initial state while *Stopped* is an intermediary state. Consideration of both final and intermediary states is necessary.

Location of Contingencies: Another implementation issue worth considering concerns the location of the contingencies. Recall that a feature consists of a *requirement(R)*, *specification(S)*, and *context(W)*. In our approach we place contingencies in the specification. This seems a natural choice as a specification consists of descriptions of *how* the properties described in the requirement will be fulfilled. Locating the contingencies in the specification has the advantage that if a state of the resource has more than one contingency, a choice can be made between the alternatives. A common factor on which such a choice can be based is cost. For example in Table 5.3 both contingencies ES5a and ES5b correspond to the case in which the DVD-R is in the *Paused-Playing* state. In this case ES5b could be a better choice as it involves fewer state changes and could be cheaper to execute than ES5a.

An alternative location of the contingencies is the context (W). The advantage of this alternative is that the specification does not have to be very specific on how the requirement is to be satisfied. It needs to only specify to the context what state is desired in the requirement. Depending on its current state, the context would select the appropriate contingency to satisfy the requirement. However, this role of the specification blurs its distinction from the requirement.

This also means that the specification is more lightweight and less complex as the responsibility of the contingency planning is shifted to the context. From a practical viewpoint, it is expected that the engineer responsible for designing a device is likely to be well versed on its behaviour and can, hence, plan better contingencies than the person who develops a specification that uses the device. However, this also removes the ability of the specification to make a choice between contingencies (in case there are two or more alternative contingencies corresponding to the same state).

7.2 Industrial Validation: Instrument Cluster Case Study

We constructed the smart home example discussed through the thesis to help us develop and illustrate the basic concepts of our approach. The example is simple enough to help us understand and motivate the problem addressed by the approach we have proposed. However, it is not enough in demonstrating the industrial relevance of our approach. For this reason, we also validated our approach by applying it to the resolution of feature interactions in an Instrument Cluster (IC) case study from DaimlerChrysler™ [Buhr *et al.* 2003].

The IC is a display device in cars consisting of various instruments which show measurement information about the status of the vehicle such as its speed, fuel level, and engine temperature. Figure 7.1 is a photo of a Instrument Cluster viewed from the driver's seat of a

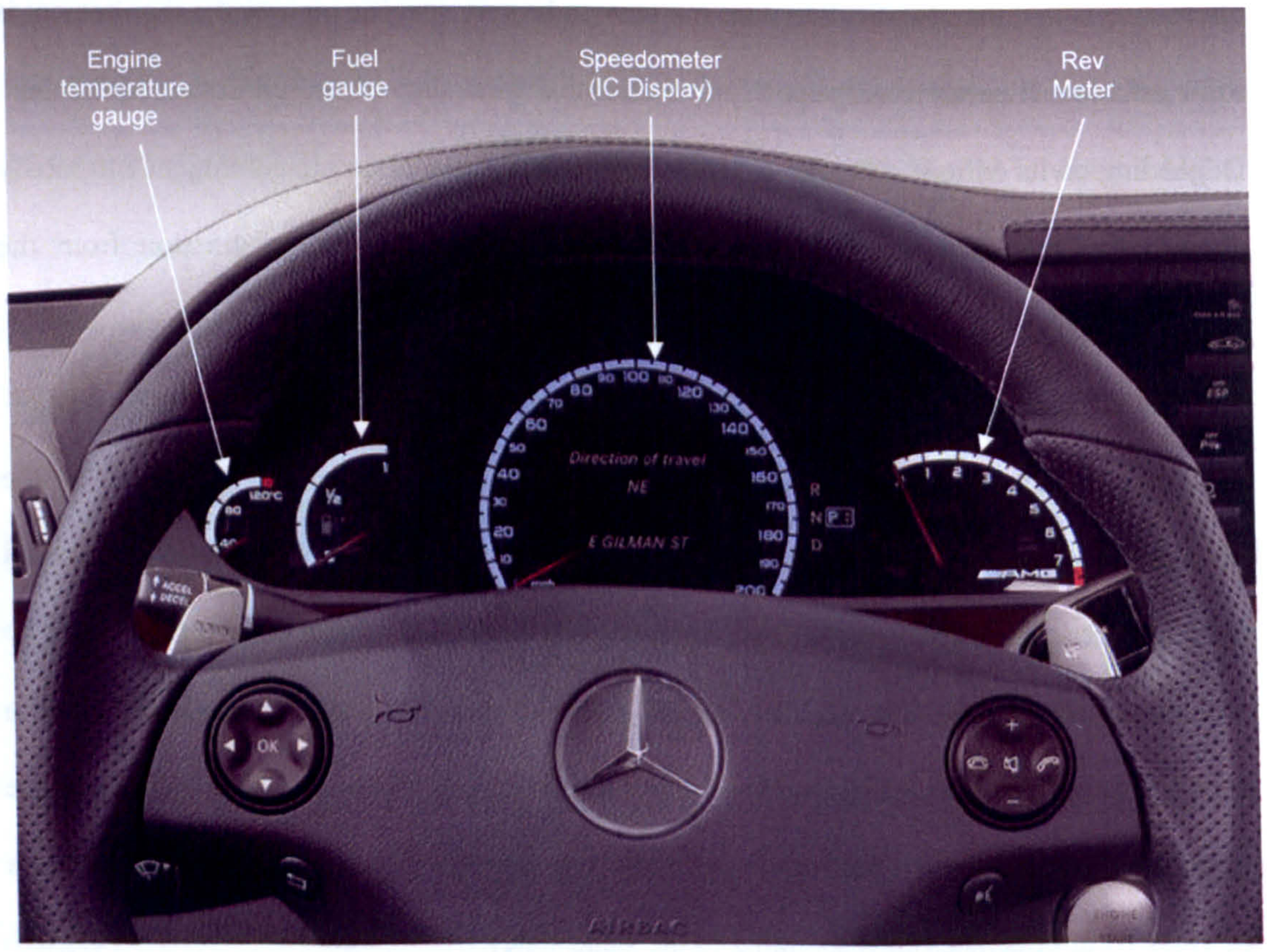


Figure 7.1 Photo of Instrument Cluster for a Mercedes S-Class S63 AMG Vehicle

Mercedes S-Class S63 AMG car. From left-to-right, it shows the engine temperature gauge, fuel gauge, speedometer, and rev meter.

This case study involved an analysis of an 83 page document which detailed the functional, hardware, and communications characteristics of the IC. Functional descriptions describe the functional behaviour of the IC including when it can be activate or deactivated, the behaviours of the rev meter, speedometer, direction indicator lights, and display. Hardware characteristics describe the physical appearance, focussing mainly on the optical design of the cluster.

Our analysis focused on activation and deactivation behavioural descriptions because this forms the larger part of the software aspect of the IC. Activation means that the cluster display lights up, while deactivation means that the display dims out and the IC is put in sleep mode. This section presents the results of applying our approach to resolving conflicts between features of the Instrument Cluster. The problem being solved is that of specifying a controller (an Activate/Deactivate machine) for activating and deactivating the IC display.

7.2.1 Requirements for Activating and Deactivating Instrument Cluster

Figure 7.2 shows the context diagram of the Activate/Deactivate machine. The context consists of the *Instrument Cluster Display*, *Ignition Lock*, *Driver's Door*, *IC Button* and *Headlights*. The required behaviour of the Activate/Deactivate machine described by seven requirements, listed in Table 7.3.

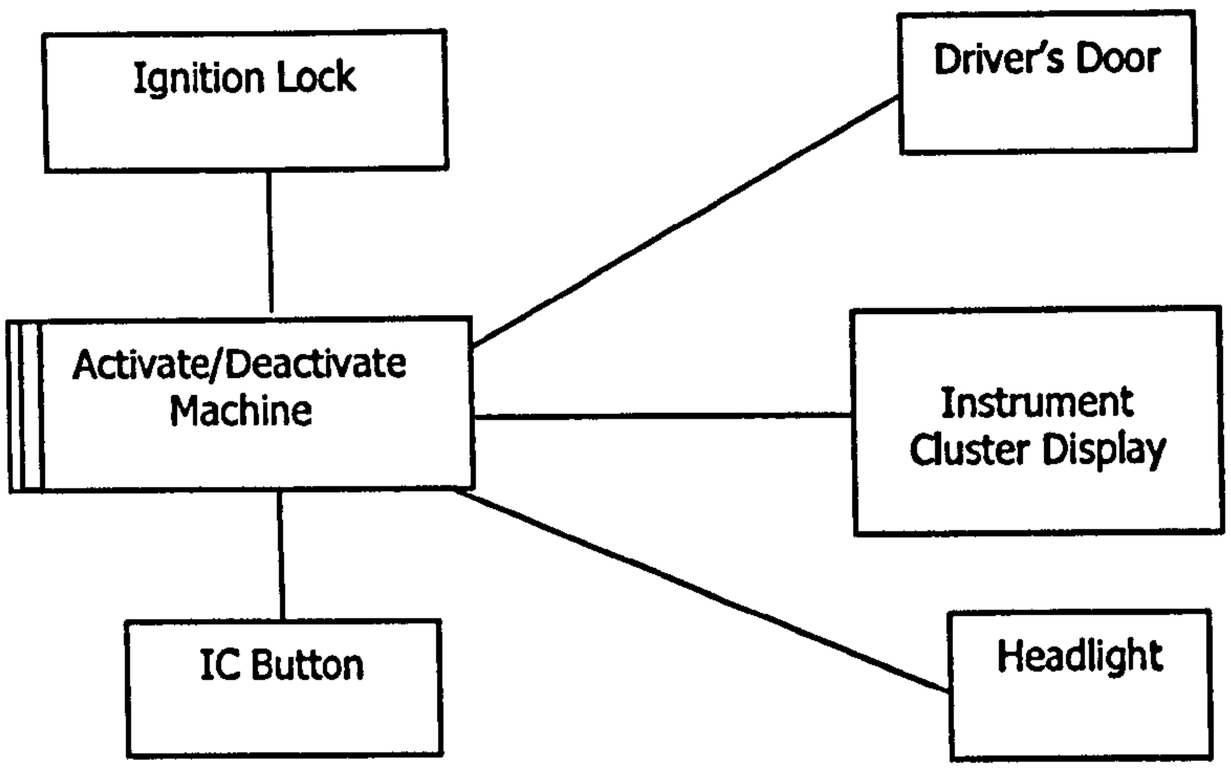


Figure 7.2 Context Diagram of Instrument Cluster Activation/Deactivation

Table 7.3 Requirements for activation/deactivation of the Instrument cluster

	Short Description	Behavioural Description
R-1	Permanent activation when ignition is on	After the ignition has been switched on the instrument cluster is activated
R-2	Permanent deactivation by switching-off ignition	Half a minute after the ignition has been switched off the instrument cluster is deactivated and all (warning) lights dim out.
R-3	Permanent activation by setting ignition key in position radio	After the ignition key is set in position radio, the instrument cluster is activated.
R-4	Temporal activation by opening the driver's door	After the driver's door has been opened the instrument cluster is activated for half a minute.
R-5	Temporal activation by closing driver's door	After the driver's door has been closed for the instrument cluster is activated for half a minute
R-6	Temporal activation by switching on headlights	After the headlights have been switched on the instrument cluster is activated for half a minute
R-7	Temporal activation with push button	After the instrument cluster push button has been applied the instrument cluster is activated for half a minute.

Whether the IC is a*ctivated* or *deactivated* depends on the status of the Ignition Lock, Driver's Door, IC Button and Headlights. For example, according to R-4 opening the Driver's Door should activate the IC for 30 seconds and deactivate it thereafter.

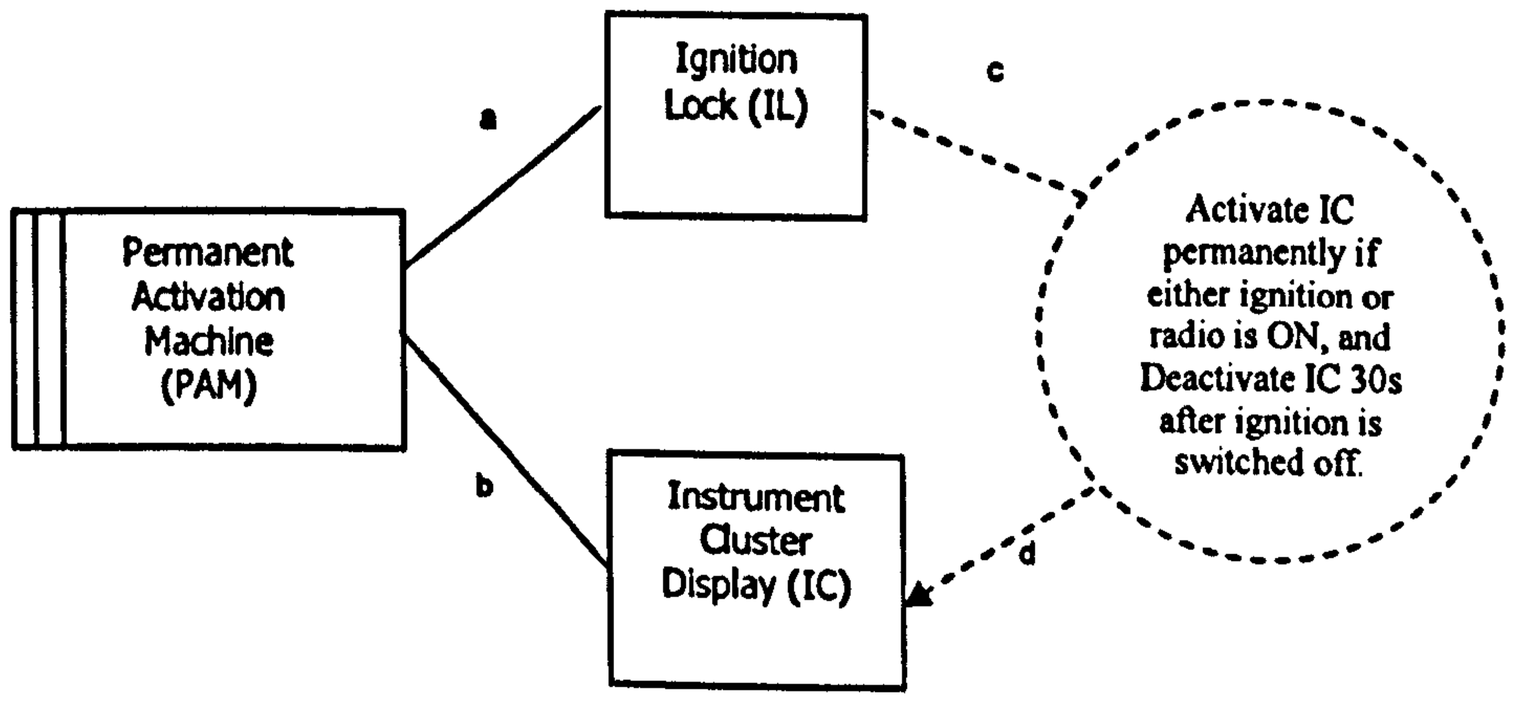
7.2.2 Features of the Instrument Cluster

Using a requirements clustering technique proposed by Hsia *et al.* [Hsia and Yaung 1988; Hsia and Gupta 1992; Hsia *et al.* 1996], we clustered these requirements into *Temporal*

Activation and *Permanent Activation* features. Temporal Activation relates to the requirements of activating the IC display for 30 seconds only based on the occurrence of a specific event. This includes requirements R-4 to R-7. Permanent Activation relates to the requirements on the IC where activation does not depend on the occurrence of intermittent events, but on the state of a monitored device. Requirements R-1 to R-3 are in this category.

For example, according to R-1 the IC should stay activated as long as the ignition is on. When the ignition is turned-off, the IC should be deactivated 30 seconds later (R-3). We use these categories in structuring the requirements into features. In this way we decompose the problem of controlling IC activation/deactivation problem into two features: *Permanent* and *Temporal Activation.*

The results of the decomposition are problem descriptions of the Permanent and Temporal Activation features shown in Figures 7.3 and 7.4, respectively.



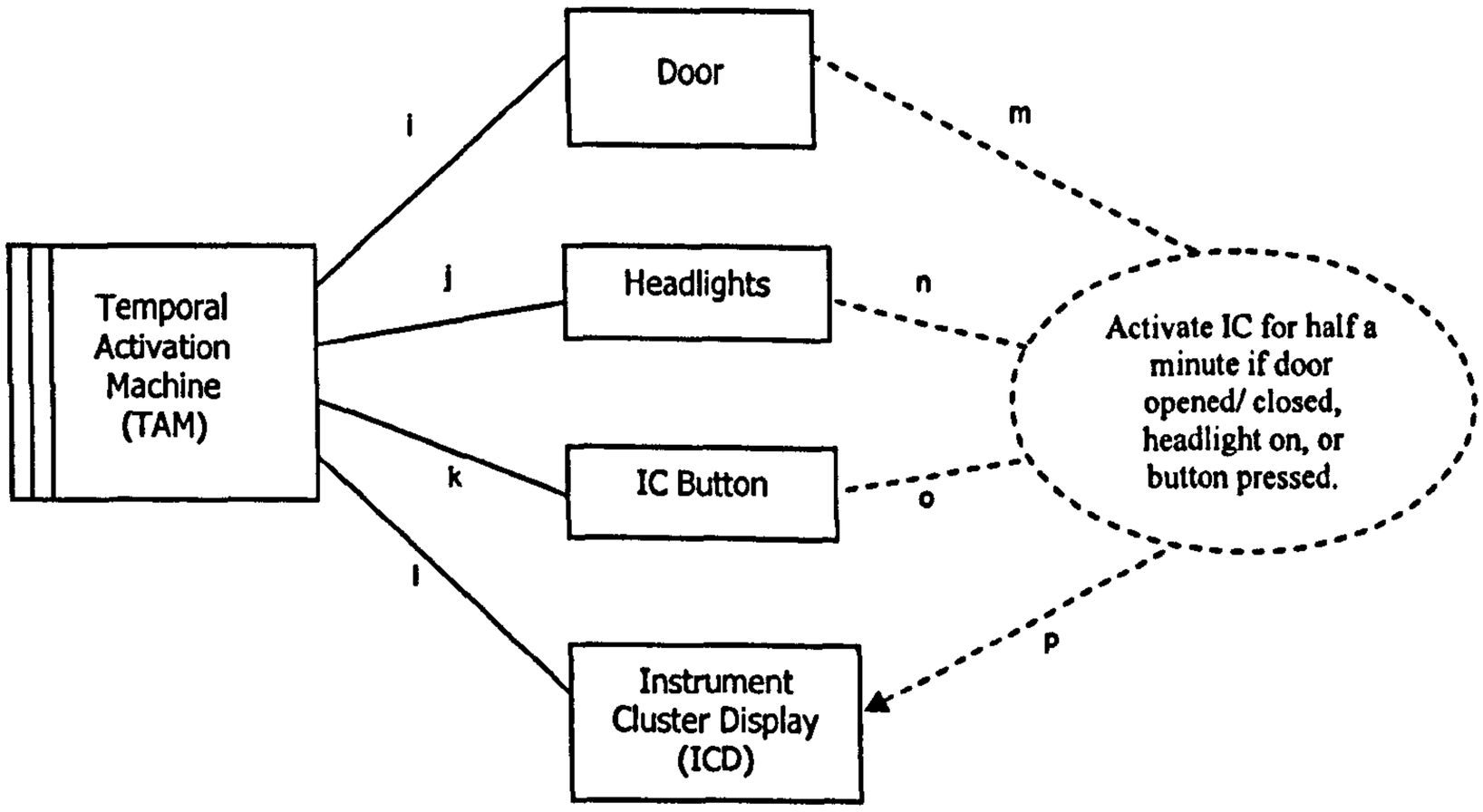

```
a: IL!{Ig_Lock=0, Ig_Lock=1, Ig_LockR=0, Ig_LockR=1}
b: PAM!{deactivate, activate}
c: IL!{IgnitionOn, IgnitionOff, RadioPowerOn,
   RadioPowerOff}
d: IC!{Deactivated, Activated}
```

Figure 7.3 Problem Diagram of Permanent Activation Feature

The requirement of the *Permanent Activation Machine (PAM)* is to *activate the IC permanently when the ignition lock is in RADIO or ON position.* When the ignition is switched off, the display is deactivated after 30 seconds. The context of the PAM consists of the *Ignition Lock* and the *Instrument Cluster* problem domain. The interface labelled *a*

consists of phenomena `Ig__Lock=0`, `Ig__Lock=1`, `Ig_LockR=0`, and `Ig_LockR=1` controlled by the Ignition Lock domain. If the ignition key is in the lock ON position then `Ig_Lock=1` (otherwise `Ig_Lock` = 0). Similarly, `Ig_LockR` = 1, if the ignition key is in the RADIO position (otherwise `Ig_LockR` = 0). The position of the key in the Ignition Lock is represented in interface c. Identifier b refers to phenomena *deactivate* and *activate*. These phenomena are the events issued by *PAM* to activate and deactivate the IC. Interface d represents the current state of the IC with phenomena Activated or Deactivated.

The requirement of the Temporal Activation feature is to *activate the Instrument Cluster for 30 seconds if either the driver's door has just been opened/closed, the headlight have just been switched-on, or the IC button has been pressed.* The context of the `Temporal Activation Machine (TAM)` includes the following problem domains: *Driver's Door*, *Headlight*, *IC Button*, and *Instrument Cluster Display*.




```
i: Door!{Status_Door_dd=0, Status_Door_dd=1}
j: Headlights!{Status_Lights=0, Status_Lights=1}
k: ICButton! {Status_IC_Button=0, Status_IC_Button=1}
l: TAM!{deactivate,activate}
m: Door! {DoorClosed, DoorOpened}
n: HeadLights!{LightsOn, LightsOff}
o: ICButton!{Pressed, NotPressed}
p: ICD! {Activated, Deactivated}
```

Figure 7.4 Problem Diagram of Temporal Activation Feature

The Temporal Activation Machine determines the current status of the driver's door through phenomena `Status_Door_dd`. If the door is observed as closed in m, then `Status_Door_dd=0`.Otherwise if `DoorOpened` is true then `Status_Door_dd=1`. The current state of the headlights is determined through interface `j`. If `LightsOn` is true in `n`, then the `Status_Lights=1` in `j`. In contrast, if `LightsOff` is true, then `Status_Lights=0`. The TAM reads the status of the ICButton through the value of phenomena `Status_IC_Button` in interface `k`. If the ICButton is `Pressed` (in interface o), then `Status_IC_Button`=1 (otherwise =0). Events to `activate` and `deactivate` the IC by the TAM are sent through interface l and in response IC Display domain is observed as either `Activated` or `Deactivated` in interface `p`.

7.2.3 Dynamic Behaviour of Instrument Cluster Display

When composed, the two features share the Instrument Cluster Display. In this section we provide domain descriptions for the display since it is the only resource shared and the likely domain where conflicts will be manifested. According to the documentation of the IC, the dynamic behaviour of the display can be described using two events and two states. The events are: *activate* and *deactivate*. The states are: `Deactivated` and `Activated`. The events and states are related by the Event Calculus domain description in Figure 7.5.

Initiates (deactivate, Deactivated, t)	[IC1]
Initiates (activate, Activated, t)	[IC2]
Terminates (deactivate, Activated, t)	[IC3]
Terminates (activate, Deactivated, t)	[IC4]

Figure 7.5 Dynamic Behaviour of Instrument Cluster

On occurrence of a *deactivate* event, the IC is *Deactivated* (IC1). IC2 states that in response to the occurrence of an *activate* event, the IC is *Activated*. IC3 and IC4 have been included for completeness.

7.2.4 Instrument Cluster Contingent Specifications

Based on the domain description of the IC, we derived contingent specifications for the features with the help of our CSG tool.

Temporal Activation Specifications: TASa (below) is the `TAM` contingent specification corresponding to the state when the IC is `Activated` . It states that on occurrence of at least one of the following events: `DoorClose, DoorOpen, ButtonPressed, HeadLightsSwitchedOn or IgnitionSwitchedOff;` if the IC is `Activated` then `deactivate` events should be prohibited for 30 seconds. After 30 seconds `TAM` should deactivate the IC.

(HoldsAt(DoorClose,t0) V HoldsAt(DoorOpen,t0) V
HoldsAt(ButtonPressed,t0) V HoldsAt(HeadLightsSwitchedOn,t0) V
HoldsAt(IgnitionSwitchedOff, t0)) Λ HoldsAt(Activated, t)
→ Happens(Prohibit(deactivate;t1;t2),t1)
Λ Happens(deactivate,t2)
Λ (t0<t1<t2<t) Λ (t2=t1+30s). [TASa]

Since the IC is `Activated`, the objective of TASa is to keep it active an then deactivate it after 30s. TASb is the `TAM` contingent specification corresponding to the state when the IC is `Deactivated`. In the case of TASb since the IC is not active, the idea is to activate it if at least one of the trigger events occurs and keep it active for 30 seconds.

(HoldsAt(DoorClose,t0) V HoldsAt(DoorOpen,t0) V HoldsAt(ButtonPressed,t0) V
HoldsAt(HeadLightsSwitchedOn,t0) V HoldsAt(IgnitionSwitchedOff, t0)) Λ
HoldsAt(Deactivated, t)
→ Happens(activate,t1) Λ Happens(Prohibit(deactivate;t1;t2),t1)

$$\Lambda \text{ Happens(deactivate,t) } \Lambda \text{ t0<t1<t2<t)}\Lambda\text{(t2=t1+30s).} \qquad \text{[TASb]}$$

Permanent Activation Specifications: PASa (below) is the contingent specification corresponding to the state when the IC is `activated`. The first part of the contingent specification states that if the ignition is `ON` (Ig_Lock=1) or in the `radio` position (`Ig_LockR`=1) then the IC should be kept activated by prohibiting a `deactivate` event.

$$\text{(HoldsAt(Ig\_Lock=1,t0) V HoldsAt(Ig\_LockR=1,t0))} \Lambda \text{ HoldsAt(Activated, t)}$$
$$\rightarrow \text{Happens(Prohibit(deactivate;t1;t),t1)} \Lambda \text{ (t0<t1<t).}$$

$$\text{HoldsAt(IgnitionSwitchedOff, t0)} \Lambda \text{ HoldsAt(Activated, t)}$$
$$\rightarrow \text{Happens(Prohibit(deactivate;t1;t2),t1)} \Lambda \text{ Happens(deactivate,t)}$$
$$\Lambda \text{ (t0<t1<t2<t)} \Lambda \text{ (t2=t1+30s).} \qquad \text{[PASa]}$$

The second part of PASa states that if the ignition has just been turned off then the IC should be deactivated 30 seconds after the occurrence of the *`IgnitionSwitchedOff`* event.

PASb describes the behaviour of the Permanent Activation Machine if the current state of the Instrument Cluster is `Deactivated`.

$$\text{(HoldsAt(Ig\_Lock=1,t0) V HoldsAt(Ig\_LockR=1,t0))} \Lambda \text{ HoldsAt(Deactivated, t0)}$$
$$\rightarrow \text{Happens(activate,t1)} \Lambda \text{ Happens(Prohibit(deactivate;t1;t),t1)} \Lambda \text{ (t0<t1<t).}$$

$$\text{HoldsAt(IgnitionSwitchedOff, t0)} \Lambda \text{ HoldsAt(Activated, t)}$$
$$\rightarrow \text{Happens(Prohibit(deactivate;t1;t2),t1)} \Lambda \text{ Happens(deactivate,t)}$$
$$\Lambda \text{ (t0<t1<t2<t)} \Lambda \text{ (t2=t1+30s).} \qquad \text{[PASb]}$$

The first part of PASb states that if the ignition key is either in the lock or radio position and the Instrument Cluster is not active then PAM should issue an `activate` event and the Composition Controller should reject `deactivate` events. The second part of PASb is identical to the second part of PASa.

7.2.5 Feature Interaction between Permanent and Temporal features

The two states of the display are mutually exclusive (only one can be true at a time). An inconsistency could occur between the two features if according to one feature the IC should be `Deactivated` while the requirement of the other feature is that it should be `Activated`. One scenario where such conflict could occur is when the car is driven at night. According to the `Permanent Activation` feature if the ignition is ON, then the IC should be `Activated`.

Meanwhile, part of the requirement of the `Temporal Activation` feature is that if the headlights are switched-on, then the IC should be temporarily `activated` for 30s. This implies that 30 seconds after the headlights have been turned-on the IC is `deactivated` – resulting in the IC display being blank! This is inconsistent with the requirement of the `Permanent Activation` feature to keep the cluster `activated` as long as the ignition is ON.

In the implementation of the features and the IC, the effect of the conflict identified above may or may not be significant. It may not be significant because the status of all the input signals are scanned every 100 milliseconds. If the `Temporal Activation` feature deactivated the IC after 30 seconds of switching-on the headlights, then the IC would be activated again by the `Permanent Activation` feature after 100ms. If the end of the 30 seconds period coincides with the beginning of the 100ms scan cycle, then (potentially) the display will be blank for 100ms.

This is illustrated in Figure 7.6. At *time = 10 ms*, TAM issues a *deactivate* event which blanks the display. If the 100ms refresh cycle begins at the same time then the display is deactivated until *time = 110ms* where it is activated again by the PAM.

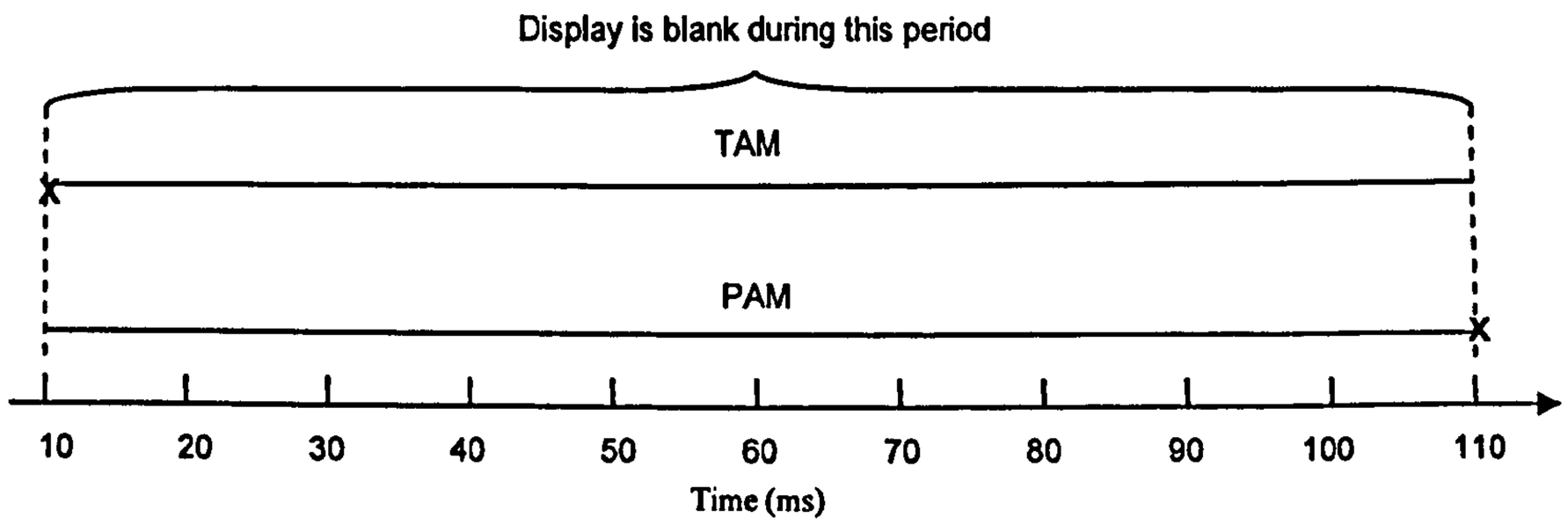


Figure 7.6 Illustration of Potential Conflict between Temporal and Permanent Activation

In the next section we explain how prioritising the features and composing them through a Composition Controller resolves the conflict discussed above.

7.2.6 Composing Specifications with a Composition Controller

Assume that the `Permanent Activation` feature has a higher priority than `Temporal Activation` and that the display is initially `Deactivated`. We now consider the behaviour of `TAM` and `PAM` when they composed through the Composition Controller. When the ignition is switched ON, `PAM` issues a `Happens(Prohibit(deactivate;t1;t)` instruction according to PASb. This instructs the Composition Controller to reject `deactivate` events as this will violate satisfaction of the `Permanent Activation` requirement. The requirement of PAM is to keep the IC active as long as the ignition is ON. The occurrence of a deactivate event violates requirement.

When the driver turns on the headlights while the ignition is ON, `TAM` issues a `Prohibit(deactivate;t1;t2)` event according to TASa (since the Instrument Cluster is already Activated by `PAM`). The Composition Controller stores the `Prohibit(deactivate;t1;t2)` in its memory for making arbitration decision later. After 30s `TAM` issues a *deactivate* event. However, this event is rejected by the Composition Controller since `PAM` ordered rejection of this event (in PASa and PASb) and it has a higher priority than `TAM`. This prevents events issued according to `TAM` specifications from having

an effect on the Instrument Cluster Display. This arbitration mechanism prevents the potential blanking of the display.

7.2.7 Validity and Implications of Case Study Results

The application of our approach to the Instrument Cluster case study has demonstrated the effectiveness of arbitration in resolving non-determinism. It highlighted a potential feature interaction between the Permanent and Temporal Activation features that could result in the Instrument Cluster display being blank for at most 100ms.

The practical implications of this problem were validated with one of the engineers responsible for specifying the IC, who accepted our conclusions based on the dynamic behaviour of the display used in the analysis. However, he also concluded that flickering of the display does not occur in practice due to the fact that the display has a transition time delay between its states. Our modelling of the dynamic behaviour of the display had not taken this into account.

The potential of flickering of the display identified by our approach highlighted that the conclusion as to whether a conflict presents a practical problem or not also depends on more specific contextual information. In the IC problem, this contextual information included the relationship between the rate at which the human eye scans the display, the rate at which input signals are scanned to update the display, and the time it takes the display to transition from being `deactivated` to `activated` (and vice versa). This emphasises the important role that context plays in the analysis of conflicts.

7.3 Chapter Summary

This chapter has presented evaluation results of our proposed approach through its application to the composition of feature in a smart home. Our approach has also been applied to the

specification of contingencies for features of an Instrument Cluster – a practical problem from the automobile domain. The evaluation results substantiates our claim that contingency planning complements arbitration by ensuring that when a non-deterministic conflict occurs the requirements of conflicting features are eventually satisfied.

Chapter 8. Conclusions and Further Work

When features are specified, the designer makes assumptions about the behaviour and initial state of the context. When the context is shared between features, assumptions about initial states are often violated. Such violation can result in requirements of some of the features not being satisfied. This thesis has argued that such violation of requirements can be avoided if specifications of features take into account that when context is shared assumptions about fixed initial states may not hold.

In addressing these initialisation concerns, we have proposed an approach that is based on contingency planning. The effectiveness of the approach has been demonstrated through an example and a case study based on a practical problem. Our evaluation of the proposed approach shows that specifications with contingencies are able to satisfy their requirements regardless of the current state of the context. We present a summary of the thesis in section 8.1, suggest pointers for further work in section 8.2, and finally, we present our conclusions in section 8.3.

8.1 Summary of Thesis Contributions

This thesis has argued that context is important in reasoning about feature interactions as conflicts manifest themselves on the context. In supporting this claim we have provided evidence from the literature in the form of feature interaction taxonomies and sources of feature interactions. Our review shows that the limitation of current approaches to feature interaction resolution is that they lack mechanisms for explicitly dealing with initialisation concerns and hence are insufficient in addressing conflicts resulting from the initialisation problem.

We have proposed an approach to the initialisation problem which combines the concepts of arbitration and contingency planning. The conceptual basis of our proposed approach has been justified by motivating how the combination of the concepts of arbitration and contingencies are relevant to feature interaction resolution. Contingency planning enables features to deal with initialisation concerns. This is achieved by equipping each feature with contingent specifications corresponding to each state of the shared resource. Depending on the current state, one of the contingencies is selected to enable a feature to satisfy its requirements.

Although contingencies deal effectively with initialisation concerns, they are insufficient in resolving non-determinism as features may still conflict as they concurrently attempt to access a shared resource. In order to resolve non-determinism we argued that arbitration is necessary in feature composition to intercede between contingent feature specifications and the shared resource.

We have demonstrated how our proposed approach can be used in practice by showing how an existing arbitration approach, the Composition Controller, can be extended with contingency planning. We identified two main steps that a development process implementing our approach could entail, namely: (1) building contingencies into specifications and (2) composing the contingent specifications through arbitration.

The first step involves deriving contingent specifications. The derivation of contingent specifications can be erroneous and time-consuming if done manually. In addressing this we presented a tool, called Contingency Specification Generator (CSG), which automates the derivation of contingent specifications. The approach have been validate through a laboratory constructed example and a case study of a practical problem.

8.2 Further Work

This thesis has argued that context sharing is the genesis of the feature interaction problem. We have identified the initialisation problem as one of the important problems that needs to be addressed in feature composition as ignoring initialisation concerns leads to bypass feature interactions. As further work on this thesis argument we identified the following research issues:

8.2.1 Problem Reduction as a source of Feature Interaction

Some software development problems are hard to solve in their original form. For convenience, it is often necessary to transform them to reduced forms that have feasible specification through *problem reduction* [Rapanotti *et al.* 2006]. However, problem reduction results in loss of the context of the original problem. As a result (of the loss of context) a specification to the reduced problem may not satisfy the original problem when the context is shared with other features (despite having been demonstrated to satisfy the original problem in isolation).

It is therefore necessary, in transforming an original problem to a reduced problem, to ensure that any resulting loss of context will not result in this type of feature interaction. To illustrate loss of context due to problem reduction as a source of feature interactions consider the following example from the automobile domain [Broy *et al.* 2007]:

> A *climate control feature* (air-conditioner) maintains the temperature inside the car at a preset value. The electrical energy that powers this air-conditioning equipment comes from the engine. The power output from the engine depends on the number of *revolutions per minute* (revs). Hence, by demanding more power from the engine, *climate control* increases the number of revs. Another common feature in luxury cars is the *automatic handbrake release feature*. This is a convenience feature that replaces traditional mechanical handle handbrakes. With this feature the driver engages and releases the handbrake by pressing a button. Once the

handbrake is engaged to drive-off the driver releases it by pressing the accelerator. When the accelerator is pressed engines revs are increased and automatic handbrake release disengages the handbrake.

One scenario in which these two features interact is if the car is driven on a hot day with the climate control feature on to maintain a cool temperature in the car. Assume the driver stops on the road side, engages electronic handbrake release with the engine running, and comes out of the car by opening the door. Since it is very hot outside, opening the door suddenly increases the temperature inside the car. In response to the temperature increase, the climate control feature suddenly demands more power from the engine – increasing the engine revs. The increase in engine revs is picked-up by the automatic handbrake release feature which disengages the handbrake. The result is that the car starts moving without the driver!

The original problem that the handbrake release feature is intended to solve is that of automatically releasing the handbrake when the driver presses the accelerator if the engine is running. Pressing the accelerator results in an increase in the engine revs. Based on this correlation, in the implementation of this feature the problem of monitoring pressing of the accelerator was transformed to that of monitoring an increase in engines revs. This transformation resulted in a subtle, yet potentially harmful feature interaction.

Problem reduction is a useful design tool as it allows a requirements analyst to reduce otherwise hard problems to more solvable forms. However, to reap the full benefits of such reductions we need a systematic way to argue and demonstrate that they are necessary, adequate, and correct. Part of our future work is: given a specification which has resulted from problem reduction, how can we formulate an argument that shows that when the context is shared, the type of feature interaction illustrated in the example above is avoided.

8.2.2 Problem Decomposition with Minimal Conflicts

In this thesis, we have modelled features as subproblems. Subproblems are a result of problem decomposition. The idea in problem decomposition is to solve each subproblem in isolation

and then compose the resulting (sub) solutions to solve the original problem. Intuitively, one would expect that since the subproblems came from the same problem, then the composition of their solutions should be trivial and provide a solution to the original problem.

However, feature interactions often arise when the solutions are (re-) composed. It is not currently known as to what extent does the way in which a problem is decomposed into subproblems contribute to conflicts that arise during composition. We envisage that knowing the relationship between problem decomposition and the resulting conflicts between features can lead to systematic guidance on how decomposition could be executed in a way that minimises feature interactions.

8.2.3 Arbitration for Distributed Resources

The basic premise of our thesis argument is that the feature interaction problem arises from sharing of resources. The approach we have proposed to resolving feature interactions assumes that the shared resources are centralised and may therefore not be applicable to distributed resources. For example the approach may not resolve the feature interaction between Originating Call Screening (OCS) and Call Forwarding Unconditional (CFU) in a telephone switching system. OCS is used to prevent a subscriber from being connected to certain pre-specified contact numbers. CFU forwards incoming calls to another subscriber. Consider a scenario with three telephone subscribers: John, Lucy and James.

Assume that James sells value added services whose content is not suitable for John's young son. In order to prevent his son from being exposed to such unsuitable material John subscribed to OCS with James on the screening list. This prevents John's son from connecting to James. If Lucy forwards all her calls to James, then dialling Lucy's number leads to the violation of John's requirements of being prevented from connecting to James. What makes this feature interaction challenging is that James, Lucy and John could belong to different administrative domains (telephone service providers).

8.2.4 Arbitration with Dynamic Priority Assignment

The arbitration approach we have used assumes that the priorities between features are fixed. The consequence of this assumption is that when a new feature is added the priorities have to be re-assigned. This need for re-assignment may not be practical for a large set of features. One approach proposed in [Zimmer and Atlee 2005] to addressing this problem is to assign priorities to classes of features rather than individual features. For example we could assign a higher priority to burglary capture features than burglar deterrence features.

However, this still raises the question of assigning priorities to features in the same class. Moreover, the feature categorisation approach still use fixed priorities assignment which are independent of system dynamics. It is worth exploring flexible priority assignment in which the assignment of priorities is a function of evolution of the feature-based application. Such an approach could aid decisions on when to give priority to a feature and when to revoke such priority.

8.2.5 Arbitration/Contingency as a Conflict Resolution Pattern

This thesis has discussed arbitration as an approach to resolving non-determinism and proposed contingency planning as an approach resolving bypass feature interactions. Based on this, a combination of the concepts of arbitration and contingency planning could be presented as a pattern for resolving non-deterministic and bypass interactions.

It is worth investigating the extent to which the combination of these concepts in the way we proposed can be generalised allowing for the resolution of similar problems across multiple domains, outside the application domains of smart homes and automotive software systems. We envisage that an application of the proposed approach to a wider and general context would further strengthen our claim that the concepts of arbitration and contingency are

complementary and highlight their limitations. This could also enable us to present their combination as a generic software design pattern for feature-based system development.

As further work, it is also worth investigating the possibility of other similar concepts (to arbitration and contingency planning) which can be applied in resolving the rest of the feature interactions characterised in current taxonomies (such as looping interaction). Combined with the approach we have proposed in this thesis, such concepts could increase the scope of the types of feature interactions that can be resolved.

8.2.6 Satisfying Failed Requirements by Retrying Rejected Events

During arbitration, events issued by a lower priority feature are rejected in favour of higher priority events. This implies that the requirements of a lower priority feature may never be satisfied unless it is retried when the resource is free. The arbitrator used in this thesis does not give feedback to the originating machine that an event has been rejected. Such feedback could give the machine the time at which it can retry a failed event. This is only possible when the exact time at which prohibition of the event will expire is known.

For example, assume that the requirement for the broadcast capture feature is to record news between 7pm and 8pm. Also assume that the broadcast capture feature has a higher priority than the burglary capture feature whose requirement is to record a burglary. Any event issued by the capture feature during the news recording period, that will violate the broadcast capture requirement will be prohibited. This means that between 7pm and 8pm the capture feature's requirement may not be satisfied. If a thief breaks-in during this period the feedback given to the burglary capture feature is that the issued event has failed and can be retried after 8pm!

However, for some events it may not be possible to know the expiry of the prohibition in advance. In the example above, assume that burglary capture has a higher priority than

broadcast capture. If a thief breaks-in at 7:35pm it is not known when he will leave and hence the duration of recording the burglary is not certain! If the requirement is that the whole episode of the burglary should be recorded then it is not possible to determine in advance how long an event from the broadcast capture feature should be prohibited. The issue is when to retry when the prohibition end of the prohibition period is not certain?

8.2.7 Termination Problems in Feature Composition

This thesis has focussed on initialisation problems in feature composition. A similar problem is termination, which concerns which state should a shared resource be left by a currently executing feature. We alluded to termination problems in section 1.4 when we mentioned stoppage concerns. The termination problem is especially important to address when there is a dependency between a currently executing feature and the next feature to use a shared resource. An abnormal termination of the current feature could leave a shared resource in a state in which it may not be use-able for the next feature as the DVD media recording example illustrated in Chapter 1. One of the research issues relevant to termination concerns is how to ensure that a shared resource is always left in a state that is use-able for the next feature. In the DVD media example, this could mean ensuring that if recording is aborted then a recording session is always closed properly.

8.2.8 Initialisation Problems in Aspect Weaving

Our work on the feature interaction problem has similarities with so-called "aspect interaction analysis" [Douence *et al.* 2004] in aspect-oriented software development [Filman *et al.* 2004]. This raises the question of whether our approach can be applied to or benefit from work in this area.

Aspect-oriented development approaches aim to address challenges associated with crosscutting concerns. Concerns are areas of interest in or focus of an application, and are the

primary criteria for decomposing software into manageable and comprehensible parts [Dijkstra 1976]. A crosscutting concern is one that spans several parts of an application, such that changing it in one part of the application requires changing it in other parts [Filman *et al.* 2004]. *Aspects* encapsulate crosscutting concerns by providing means for their systematic identification, isolation and modularisation, representation, and composition [Rashid *et al.* 2002; Rashid *et al.* 2003; Baniassad *et al.* 2006]. Within aspect-oriented development, our work is more relevant to "early aspects" [Amirat *et al.* 2006; Baniassad *et al.* 2006; Weston *et al.* 2008]. The work on early aspects advocates the identification and modularisation of crosscutting concerns at requirements engineering time [Nuseibeh 2004; Rashid *et al.* 2004].

Aspects are composed by weaving them through a base concern, which often changes the behaviour of that base concern. Aspect weaving is the composition process that integrates aspects with base concerns in an application and a base concern is a non-crosscutting concern through which aspects are composed [Rashid *et al.* 2004; Weston *et al.* 2008]. If the aspects being composed are conflicting, their composition may result in behaviour that does not satisfy the requirements satisfied by each aspect in isolation - introducing subtle and undesirable interactions among aspects [Falcarin and Torchiano 2006]. In this respect, we can consider aspect interaction as a special case of the feature interaction problem and aspect weaving as a special case of feature composition.

Treating aspect interaction as a special case of feature interaction means that the base concern modified by the application of aspects can be considered as a shared resource. This suggests that there may be scope for applying our approach to aspect interaction analysis or for our work to benefit from the work on early aspects interaction analysis. However, before this is possible some research issues need to be addressed. In particular, how can our initialisation problem be described for aspect composition? In deriving contingent specifications we use domain descriptions of shared resources. We have used domain descriptions of physical devices, and applying our approach to aspect composition would need a domain description of

base concerns. We described our domain descriptions in terms of states and events. It is therefore worth investigating what are the equivalent states and events in base concerns, and whether states and events are sufficient abstractions for documenting domain descriptions of these concerns.

Our technique for deriving contingent specifications assumes that the behaviour of the shared resource does not change. In aspect composition this assumption does not hold as the behaviour of a base concern changes when an aspect is applied. Hence, deriving contingent aspect specifications should take into account the changing behaviour of the shared resource. This means that contingent specifications would have to be derived each time there is a change in the behaviour of a base concern. It is worth investigating how our approach copes with shared resources of potentially varying behaviour.

8.3 Conclusion

The feature interaction problem arises from context sharing. A common approach to resolving conflicts between features contesting for a resource is arbitration. This approach resolves conflicts by assigning priorities to conflicting features such that a higher priority feature gains access to the resource. However, this approach does not guarantee that the requirement of the feature that eventually gains access to the resource will be satisfied. This is because the current state of the resource may not match its pre-conditions due to the state having been changed by features that used the resource earlier. We characterised this as the initialisation problem.

This thesis has proposed an approach to the initialisation problem. Our approach is based on the concept of contingency planning and involves analysing initialisation concerns in the problem space. The analysis of initialisation concerns enables a requirements analyst to consider all possible states of the context and derive multiple contingent specifications. The

selection of a specific contingent specification helps avoid initialisation concerns associated with each state of the context.

We have demonstrated its relevance to the resolution of bypass feature interactions by combing it with arbitration. The evaluation results of our proposed approach suggest that contingency planning ensures that, in the event that the state of the resource in a model is inconsistent with the actual state, the requirements of conflicting features are eventually satisfied (where they can be satisfied).

APPENDIX 1 -- DVD-R Domain Descriptions encoded in ECharts

```
public machine DVDR {
        <* final private ExternalPort p1 *>
        <* private String currentState = "STOPPED" *>

        public DVDR(ExternalPort p1) {
                this.p1 = p1;
        }
        initial state STOPPED;
        state PLAYING;
        state PAUSED_PLAYING;
        state RECORDING;
        state PAUSED_RECORDING;

transition STOPPED - p1 ? String [message == "play" && currentState == "STOPPED"] / {
                        System.out.println("Current State: PLAYING");
                        currentState = "PLAYING";
                } -> PLAYING;

transition PLAYING - p1 ? String [message == "stop"] / {
                        System.out.println("Current State: STOPPED");
                        currentState = "STOPPED";
                } -> STOPPED;

transition PLAYING - p1 ? String [message == "pause" && currentState == "PLAYING"] / {
                        System.out.println("Current State: PAUSED_PLAYING");
                        currentState = "PAUSED_PLAYING";
                } -> PAUSED_PLAYING;

transition PAUSED_PLAYING - p1 ? String [message == "play" && currentState == "PAUSED_PLAYING"]
                / {
                        System.out.println("Current State: PLAYING");
                        currentState = "PLAYING";
                } -> PLAYING;

transition PAUSED_PLAYING - p1 ? String [message == "stop" && currentState == "PAUSED_PLAYING"]
                / {
                        System.out.println("Current State: STOPPED");
                        currentState = "STOPPED";
                } -> STOPPED;

transition STOPPED - p1 ? String [message == "record" && currentState == "STOPPED"] / {
                        System.out.println("Current State: RECORDING");
                        currentState = "RECORDING";
                } -> RECORDING;

transition RECORDING - p1 ? String [message == "stop"] / {
                        System.out.println("Current State: STOPPED");
                        currentState = "STOPPED";
                } -> STOPPED;

transition RECORDING - p1 ? String [message == "pause" && currentState == "RECORDING"] / {
                        System.out.println("Current State: PAUSED_RECORDING");
                        currentState = "PAUSED_RECORDING";
                } -> PAUSED_RECORDING;

transition PAUSED_RECORDING - p1 ? String [message == "record" && currentState ==
                "PAUSED_RECORDING"] / {
                        System.out.println("Current State: RECORDING");
                        currentState = "RECORDING";
                } -> RECORDING;

transition PAUSED_RECORDING - p1 ? String [message == "stop"] / {
                        System.out.println("Current State: STOPPED");
                        currentState = "STOPPED";
                } -> STOPPED;

transition STOPPED - p1 ? String [message == "pause"] / {
                        currentState = "STOPPED";
                } -> STOPPED;

transition STOPPED - p1 ? String [message == "stop"] / {
                        currentState = "STOPPED";
                } -> STOPPED;
```

```
transition RECORDING - p1 ? String [message == "play"] / {
                                        currentState = "RECORDING";
                                    } -> RECORDING;

transition RECORDING - p1 ? String [message == "record"] / {
                                        currentState = "RECORDING";
                                    } -> RECORDING;

transition PAUSED_RECORDING - p1 ? String [message == "pause"] / {
                                        currentState = "PAUSED_RECORDING";
                                    } -> PAUSED_RECORDING;

transition PAUSED_RECORDING - p1 ? String [message == "play"] / {
                                        currentState = "PAUSED_RECORDING";
                                    } -> PAUSED_RECORDING;

    transition PLAYING - p1 ? String [message == "play"] / {
                                        currentState = "PLAYING";
                                    } -> PLAYING;

    transition PLAYING - p1 ? String [message == "record"] / {
                                        currentState = "PLAYING";
                                    } -> PLAYING;

    transition PAUSED_PLAYING - p1 ? String [message == "record"] / {
                                        currentState = "PAUSED_PLAYING";
                                    } -> PAUSED_PLAYING;

    transition PAUSED_PLAYING - p1 ? String [message == "pause"] / {
                                        currentState = "PAUSED_PLAYING";
                                    } -> PAUSED_PLAYING;

}
```

Bibliography

Abadi, M. and L. Lamport (1993). "Composing specifications." ACM Transactions on Programming Languages and Systems (TOPLAS) 15(1): pp. 73-132.

Accorsi, R., C. Areces, W. Bouma and M. d. Rijke (2000). Features as Constraints. Feature Interactions in Telecommunications and Software Systems. Amsterdam, IOS Press: pp. 210-225.

Akyildiz, I. F., H. Rudin, L. G. Bouma, N. Griffeth and K. Kimbler (2000). "Special issue on the feature interactions in telecommunications systems." Computer Networks 32(4): pp.

Albert, K., K. Jensen and R. Shapiro (1989). "A Tool Package Supporting the Use of Colored Nets." Petri Net Newsletter 32: pp. 22-35.

Amirat, A., D. Meslati and M. T. Laskri (2006). Elicitation of crosscutting aspects at the early phases of software development. Information and Communication Technologies: pp. 3575-3576.

Amyot, D. (2001). Use Case Maps as a Feature Description Notation. Language Constructs for Describing Features. Berlin, Springer: pp.

Amyot, D. and A. Eberlein (2003). "An Evaluation of Scenario Notations and Construction Approaches for Telecommunication Systems Development." Telecommunications Systems Journal 24(1): pp. 61-94.

Amyot, D., T. Gray, R. Liscano, L. Logrippo and J. Sincennes (2005). "Interactive Conflict Detection and Resolution for Personalized Features." Journal of Communications and Networks 7(3): pp. 1-14.

Amyot, D. and L. Logrippo (2004). "Special issue: Directions in feature interaction research." Computer Networks 45(5): pp.

Anderson, A. and L. Bambrick (2007). Air Crash Investigation Special: Who's Fly the Plane. Air Crash Investigation Special. Canada, Cineflix International: pp. 1 Hour.

Areces, C., W. Bouma and M. d. Rijke (2000). Feature Interaction as a Satisfiability Problem. Proceedings of the 7th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems: pp. 339.

Bandara, A. K., E. C. Lupu and A. Russo (2003). Using event calculus to formalise policy specification and analysis. Proceedings of the 4th IEEE International Workshop on Policies for Distributed Systems and Networks: pp. 26-39.

Baniassad, E., P. C. Clements, J. Araujo, A. Moreira, A. Rashid and B. Tekinerdogan (2006). "Discovering early aspects." IEEE Software 23(1): pp. 61-70.

Bisbal, J. and B. H. C. Cheng (2004). "Resource-based Approach to Feature Interaction in Adaptive Software." Proceedings of the 1st ACM SIGSOFT workshop on Self-managed systems: pp. 23 - 27.

Blair, L., T. Jones and S. Reiff-Marganiec (2002). A feature manager approach to the analysis of component-interactions. Proceedings of the Fifth International Conference on Formal Methods for Open Object-Based Distributed Systems, Kluwer, The Netherlands: pp. 233 - 248.

Blair, L. and K. J. Turner (2005). Handling Policy Conflicts in Call Control. Proc. International Conference on Feature Interaction VIII, Amsterdam, IOS Press: pp.

Bond, G. W. An Introduction to ECharts: The Concise User Manual: http://echarts.org/index.php; Accessed on

Bond, G. W., E. Cheung, K. H. Purdy, P. Zave and C. Ramming (2004). "An Open Architecture for Next-Generation Telecommunication Services." ACM Transactions on Internet Technology (TOIT) 4(1): pp. 83-123.

Bonn, R. L. (1972). " Arbitration: An Alternative System for Handling Contract Related Disputes." Administrative Science Quarterly 17(2): pp. 254-264.

Bordeaux, L., Y. Hamadi and L. Zhang (2006). "Propositional Satisfiability and Constraint Programming: A comparative survey." ACM Computing Surveys 38(4): pp. 12.

Braithwaite, K. H. and J. M. Atlee (1994). Towards automated detection of feature interactions. Feature Interactions in Telecommunications Systems, IOS Press: pp. 36-59.

Brandin, B. A. and W. M. Wonham (1994). "Supervisory control of timed discrete-event systems." IEEE Transactions on Automatic Control 39(2): pp. 329-342.

Bredereke, J. (2004). On Feature Orientation and on Requirements Encapsulation Using Families of Requirements. Objects, Agents, and Features. Berlin Heidelberg, Springer-Verlag. Volume 2975 / 2004: pp. 26–44.

Bredereke, J. (2005). Configuring Members of a Family of Requirements Using Features. Feature Interactions in Telecommunications and Software Systems VIII, Leister, U.K., IOS Press: pp. 96-113.

Broy, M., I. H. Kruger, A. Pretschner and C. Salzmann (2007). "Engineering Automotive Software." Proceedings of the IEEE 95(2): pp. 356-373.

Buhr, K., N. Heumesser, F. Houdek, H. Omasreiter, F. Rothernel, R. Tavakoli and T. Zink (2003). DaimlerChrysler Demonstrator: System Specification Instrument Cluster: http://www.empress-itea.org/deliverables/D5.1_Appendix_B_v1.0_Public_Version.pdf; Accessed on 23-05-2008

Calder, M., M. Kolberg, E. Magill and S. Reiff-Marganiec (2003). "Feature interaction: A critical review and considered forecast." Computer Networks 41(1): pp. 115-141.

Calder, M. and E. Magill (2000). Feature Interactions in Telecommunications and Software Systems VI. Amsterdam, The Netherlands, IOS Press.

Calder, M. and A. Miller (2001). Using SPIN for Feature Interaction Analysis - A Case Study. Proceedings of the 8th international SPIN workshop on Model checking of software, Toronto, Ontario, Canada, Springer-Verlag New York, Inc.: pp. 143-162.

Calder, M. and A. Miller (2006). "Feature interaction detection by pairwise analysis of LTL properties: a case study." Formal Methods in System Design 28(3): pp. 213-261.

Cameron, E. J., N. Griffeth, Y.-J. Lin, M. E. Nilson, W. K. Schnure and H. Velthuijsen (1993). "A feature-interaction benchmark for IN and beyond." IEEE Communications Magazine 31(3): pp. 64-69.

Cameron, E. J. and H. Velthuijsen (1993). "Feature interactions in telecommunications systems." IEEE Communications Magazine 31(8): pp. 18-23.

Charbonnier, F., H. Alla and R. David (1999). "Discrete-event dynamic systems." IEEE Transactions on Control Systems Technology 7(2): pp. 175-187.

Chen, K., W. Zhang, H. Zhao and H. Mei (2005). An Approach to Constructing Feature Models Based on Requirements Clustering. 13th IEEE International Conference on Requirements Engineering (RE'05): pp. 31-40.

Chen, Y.-L., S. Lafortune and F. Lin (1995). Modular Supervisory Control with Priorities for Discrete Event Systems. Proceedings of the 34th IEEE Conference on Decision and Control.: pp. 409-415.

Chi, C. and R. Hao (2007). "Test generation for interaction detection in feature-rich communication systems." Journal of Computer Networks: Special Issue on Feature Interaction 51(2): pp. 426-438.

Classen, A., P. Heymans and P.-Y. Schobbens (2008). What's in a Feature : A Requirements Engineering Perspective. Fundamental Approaches to Software Engineering: pp. 16-30.

Conry, S. E., K. Kuwabara, V. R. Lesser and R. A. Meyer (1991). "Multistage negotiation for distributed constraint satisfaction." Systems, Man and Cybernetics, IEEE Transactions on 21(6): pp. 1462-1477.

Cortellessa, V., B. Cukic, A. Mili, M. Shereshevsky, H. Sandhu, D. Del and M. Napolitano (2000). Certifying Adaptive Flight Control Software. Proceedings of the ISACC2000 - The Software Risk Management Conference, Reston, VA, USA: pp.

Dahlstedt, A. G. and A. Persson (2003). Requirements Interdependencies - Moulding the State of Research into a Research Agenda. Proceedings of the 9th International Workshop on Requirements Engineering: Foundations for Software Quality, Klagenfurt/Velden, Autria: pp. 55-64.

Damas, C., B. Lambeau, P. Dupont and A. van Lamsweerde (2005). "Generating annotated behavior models from end-user scenarios." IEEE Transactions on Software Engineering 31(12): pp. 1056 - 1073.

Dijkstra, E. W. (1976). A Discipline of Programming. Michigan, USA, Prentice Hall.

Dini, P., A. Clemm, T. Gray, F. J. Lin, L. Logrippo and S. Reiff-Marganiec (2004). "Policy-enabled mechanisms for feature interactions: reality, expectations, challenges." Computer Networks 45(5): pp. 585-603.

Dolev, S. and Y. A. Haviv (2006). "Self-stabilizing microprocessor: analyzing and overcoming soft errors." IEEE Transactions on Computers 55(4): pp. 385-399.

Dolev, S. and R. Yagel (2008). "Towards Self-Stabilizing Operating Systems." IEEE Transactions on Software Engineering 34(4): pp. 564-576.

Douence, R., P. Fradet and M. Sudholt (2004). Composition, reuse and interaction analysis of stateful aspects. Proceedings of the 3rd international conference on Aspect-oriented software development. Lancaster, UK, ACM: pp. 141-150.

Dunlop, N., J. Indulska and K. Raymond (2003). Methods for conflict resolution in policy-based management systems. Proceedings of the 7th IEEE International Conference on Enterprise Distributed Object Computing: pp. 98-109.

Easterbrook, S. (1993). Domain modelling with hierarchies of alternative viewpoints. Proceedings of the IEEE International Symposium on Requirements Engineering, 1993, San Diego, CA, USA: pp. 65-72.

Easterbrook, S. M. and B. A. Nuseibeh (1996). "Using ViewPoints for Inconsistency Management." Software Engineering Journal 11(1): pp.

Elfe, C. D., E. C. Freuder and D. Lesaint (1998). "Dynamic constraint satisfaction for feature interaction." BT Technology Journal 16(3): pp.

Falcarin, P. and M. Torchiano (2006). Automated Reasoning on Aspects Interactions. 21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06), Tokyo, Japan: pp. 313-316.

Felty, A. P. and K. S. Namjoshi (2003). "Feature specification and automated conflict detection." ACM Transactions on Software Engineering and Methodology (TOSEM) 12(1): pp. 3 - 27.

Ferber, S., J. Haag and J. Savolainen (2002). Feature Interaction and Dependencies: Modeling Features for Reengineering a Legacy Product Line. Software Product Lines : Second International Conference, SPLC 2, August 19-22, 2002. Proceedings, San Diego, CA, USA,, Springer-Verlag GmbH: pp. 235.

Fickas, S. and M. S. Feather (1995). Requirements monitoring in dynamic environments. *Proceedings* of the Second IEEE International Symposium on Requirements Engineering: pp. 140 - 147.

Filman, R. E., T. Elrad, S. Clarke and M. Aksit (2004). Aspect-Oriented Software Development. London, United Kingdom, Addison-Wesley.

Fu, Q., P. Hamois, L. Logrippo and J. Sincennes (2000). "Feature interaction detection: a LOTOS-based approach." Computer Networks 32(4): pp. 433-448.

Gelfond, M. and V. Lifschitz (1993). "Representing action and change by logic programs." The Journal of Logic Programming 17(2-4): pp. 301-321.

Giannakopoulou, D. and J. Magee (2003). Fluent model checking for event-based systems. Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering. Helsinki, Finland, ACM Press: pp. 257-266.

Gibson, J. P. (1997). Feature requirements models: Understanding interactions. Feature Interactions in Telecommunications Networks IV. Montréal, Canada, IOS Press. 90-5199-347-1: pp. 46--60.

Gibson, J. P., G. Hamilton and D. Méry (1999). Integration Problems in Telephone Feature Requirements. Proceedings of the 1st International Conference on Integrated Formal Methods, Springer-Verlag London, UK: pp. 129-148.

Godskesen, J. C. (1995). A Formal Framework for Feature Interaction with Emphasis on Testing. Feature Interactions in Telecommunications Systems III, IOS Press: pp. 21- 30.

Gorse, N., L. Logrippo and J. Sincennes (2006). "Formal Detection of Feature Interactions with Logic Programming and LOTOS." Jornal of Software and Systems Modeling 5(2): pp. 135.

Gould, R. (1988). Graph Theory. California, Addison-Wesley.

Haley, C. B., R. Laney, J. D. Moffett and B. Nuseibeh (2008). "Security Requirements Engineering: A Framework for Representation and Analysis." IEEE Transactions on Software Engineering 34(1): pp. 133-153.

Hall, R. J. (2000a). "Feature combination and interaction detection via foreground/background models." Computer Networks 32(4): pp. 449-469.

Hall, R. J. (2000b). Feature Interaction in Electronic Mail. Feature Interactions in Telecommunications and Software Systems VI. Glasgow, Scotland, UK, IOS Press: pp.

Hall, R. J. (2005). "Fundamental Nonmodularity in Electronic Mail." Automated Software Engineering 12(1): pp. 41-79.

Hamed, H. and E. Al-Shaer (2006). "Taxonomy of conflicts in network security policies." Communications Magazine, IEEE 44(3): pp. 134-141.

Hay, J. and J. Atlee (2000). "Composing Features and Resolving Interactions." ACM SIGSOFT Software Engineering Notes Volume 25(Issue 6): pp. 110-119.

Hsi, I. and C. Potts (2000). Studying the Evolution and Enhancement of Software Features. 16th IEEE International Conference on Software Maintenance (ICSM'00): pp. 143-151.

Hsia, P. and A. Gupta (1992). Incremental delivery using abstract data types and requirements clustering. Proceedings of the Second International Conference on Systems Integration (ICSI '92): pp. 137-150.

Hsia, P., C. T. Hsu, D. C. Kung and L. B. Holder (1996). User-Centered System Decomposition: Z-Based Requirements Clustering. Proceedings of the Second International Conference on Requirements Engineering (ICRE'96): pp. 126.

Hsia, P. and A. T. Yaung (1988). Another approach to system decomposition: requirements clustering. Proceedings of the 12th International Computer Software and Applications Conference (COMPSAC 88): pp. 75-82.

Hwu, W.-M. W. and T. M. Conte (1994). "The susceptibility of programs to context switching." IEEE Transactions on Computers 43(9): pp. 994-1003.

Jackson, M. (2001). Problem frames : analysing and structuring software development problems. Harlow, Addison-Wesley, 2001.

Jackson, M. and P. Zave (1993). Domain descriptions. Proceedings of IEEE International Symposium on Requirements Engineering: pp. 56-64.

Jackson, M. and P. Zave (1998). "Distributed Feature Composition: A Virtual Architecture for Telecommunications Services." IEEE Transactions on Software Engineering 24(10): pp. 831-847.

Jia, Y. and J. M. Atlee (2004). "Run-Time Management of Feature Interactions." 6th ICSE Workshop on Component-Based Software Engineering 29(4): pp.

Kaindl, H. (2005). "A scenario-based approach for requirements engineering: Experience in a telecommunication software development project." Systems Engineering 8(3): pp. 197-210.

Kang, K. C., S. Kima, J. Lee, K. Kim, E. Shin and M. Huh (1998). "FORM: A feature-;oriented reuse method with domain-;specific reference architectures." Annals of Software Engineering 5(0): pp. 143 - 168.

Keck, D. O. and P. J. Kuehn (1998). "The Feature and Service Interaction Problem in Telecommunications Systems: A Survey." IEEE Transactions on Software Engineering 24(10): pp. 779 - 796.

Kolberg, M., E. Magill, D. Marples and S. Tsang (2001). Feature Interactions in Services for Internet Personal Appliances. In Proceedings of IEEE International Conference on Communications (ICC-2002), New York: pp. 2613-2618.

Kolberg, M. and E. H. Magill (2007). "Managing feature interactions between distributed SIP call control services." Journal of Computer Networks: Special Issue on Feature Interaction 51(2): pp. 536-557.

Kolberg, M., E. H. Magill and M. Wilson (2003). "Compatibility Issues between Services Supporting Networked Appliances." IEEE Communications Magazine 41(11): pp. 136-147.

Krebs, B. (2008). Cyber Incident Blamed for Nuclear Power Plant Shutdown: http://www.washingtonpost.com/wp-dyn/content/article/2008/06/05/AR2008060501958.html; Accessed on 04-12-2008

Kryvyi, S. L. and L. Y. Matveyeva (2003). "Formal Methods of Analysis of System Properties." Journal of Cybernetics and Systems Analysis 39(2): pp. 174 - 191.

Laney, R., L. Barroca, M. Jackson and B. Nuseibeh (2004). Composing Requirements Using Problem Frames. 12th IEEE International Requirements Engineering Conference (RE'04): pp. 122-131.

Laney, R., M. Jackson and B. Nuseibeh (2005). Composing Problems: Deriving specifications from inconsistent requirements. The Open University, Milton Keynes, U.K.

Laney, R., T. T. Tun, M. Jackson and B. Nuseibeh (2007). Composing Features by Managing Inconsistent Requirements. 9th International Conference on Feature Interactions in Software and Communication Systems, Grenoble, France: pp.

Lin, F. J. and Y.-J. LIN (1994). A building block approach to detecting and resolving feature interaction. The Second Int'l Workshop on Feature Interactions in Telecommunications Software Systems,, Amsterdam, The Netherlands: pp.

Liu, X., H. Yang and H. Zedan (1997). Formal methods for the re-engineering of computing systems: a comparison. Computer Software and Applications Conference, 1997. COMPSAC '97. Proceedings., The Twenty-First Annual International: pp. 409-414.

Logrippo, L. (1998). "Special issue on feature interactions in telecommunications software." Computer Networks and ISDN Systems 30(15): pp.

Lorentsen, L., A.-P. Tuovinen and J. Xu (2002). Modelling Feature Interaction Patterns in Nokia Mobila Phones using Coloured Petri Nets. 23th International Conference on Application and Theory of Petri Nets, Adelaide, Australia, Springer-Verlag Berlin Heidelberg: pp. 294-313.

Lu, Y., G. Wei and T.-Y. Cheung (2001). Managing feature interactions in telecommunications systems by Temporal Colored Petri nets. Proceedings of the Seventh IEEE International Conference on Engineering of Complex Computer Systems, 2001., Skovde, Sweden: pp. 260-269.

Lupu, E. C. and M. Sloman (1999). "Conflicts in policy-based distributed systems management." Software Engineering, IEEE Transactions on 25(6): pp. 852-869.

Maccari, A. and A. Heie (2005). "Managing infinite variability in mobile terminal software." Software: Practice and Experience 35(6): pp. 513-537.

McKinley, P. K., S. M. Sadjadi, E. P. Kasten and B. H. C. Cheng (2004). "Composing Adaptive Software." IEEE Computer 37(7): pp. 56-64.

Metzger, A. (2004). "Feature interactions in embedded control systems." Computer Networks 45(5): pp. 625-644.

Metzger, A. and C. Webel (2003). Feature Interaction Detection in Building Control Systems by Means of a Formal Product Model. Feature Interactions in Telecommunications and Software Systems VII, Ottawa, Canada, IO Press: pp. 105-121.

Mitra, S., P. Sanda and N. Seifert (2007). Soft Errors: Technology Trends, System Effects, and Protection Techniques. Proceedings of the 13th IEEE International On-Line Testing Symposium: pp. 4.

Mueller, E. T. (2006a). Commonsense Reasoning. San Francisco, Morgan Kaufmann.

Mueller, E. T. (2006b). "Event calculus and temporal action logics compared." Artificial Intelligence 170(11): pp. 1017-1029.

Nakamura, M., H. Igaki and K.-i. Matsumoto (2004a). Feature Interactions in Integrated Services of Networked Home Appliances: An Object Oriented Approach. 8th International Conference on Feature Interactions in Telecommunications and Software Systems, Leicester, UK: pp.

Nakamura, M., T. Kikuno, J. Hassine and L. Logrippo (2000). Feature Interaction Filtering with Use Case Maps at Requirements Stage. Feature Interactions in Telecommunications and Software Systems VI, IOS Press: pp.

Nakamura, M., P. Leelaprute and T. Kikuno (2002). Deriving Interaction-Prone Scenarios in Feature Interaction Filtering with Use Case Maps. Proceedings of the Seventh IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS'02): pp. 237 - 244.

Nakamura, M., P. Leelaprute, K.-i. Matsumoto and T. Kikuno (2004b). "On detecting feature interactions in the programmable service environment of Internet telephony." Computer Networks 45(5): pp. 605-624.

Nhlabatsi, A., R. Laney and B. Nuseibeh (2008). "Feature Interaction: the Security Threat from within Software Systems." Progress in Informatics(5): pp. 75-89.

Nuseibeh, B. (2004). Crosscutting requirements. Proceedings of the 3rd international conference on Aspect-oriented software development. Lancaster, UK, ACM: pp. 3-4.

Nuth, P. R. and W. J. Dally (1991). A mechanism for efficient context switching. Proceedings of IEEE International Conference on
Computer Design: pp. 301-304.

Oreizy, P., M. M. Gorlick, R. N. Taylor, D. Heimbigner, G. Johnson, N. Medvidovic, A. Quilici, D. S. Rosenblum and A. L. Wolf (1999). "An Architecture-Based Approach to Self-Adaptive Software." IEEE Intelligent Systems and Their Applications 14(3): pp. 54 - 62.

Palmer, S. R. and J. M. Felsing (2002). A Practical Guide to Feature-Driven Development, Pearson Education.

Pang, J. and L. Blair (2002). "An Adaptive Run Time Manager for the Dynamic Integration and Interaction Resolution of Features." 22nd International Conference on Distributed Computing Systems Workshops (ICDCSW '02): pp. 445 - 450.

Pomakis, K. P. and J. M. Atlee (1996). Reachability analysis of feature interactions: a progress report. Proceedings of the 1996 ACM SIGSOFT international symposium on Software testing and analysis, San Diego, California, United States, ACM Press New York, NY, USA: pp. 216 - 223.

Pulvermueller, E., A. Speck, J. O. Coplien, M. D'Hondt and W. D. Meuter (2002). Feature Interaction in Composed Systems. Object-Oriented Technology, Springer Berlin / Heidelberg: pp. 1-16.

Rapanotti, L., J. G. Hall and Z. Li (2006). "Deriving specifications from requirements through problem reduction." IEE Proceedings - Software 153(5): pp. 183-198.

Rashid, A., A. Moreira and J. Araujo (2003). Modularisation and composition of aspectual requirements. Proceedings of the 2nd International Conference on Aspect-Oriented Software Development. Boston, Massachusetts, ACM: pp. 11-20.

Rashid, A., A. Moreira and B. Tekinerdogan (2004). "Early aspects: aspect-oriented requirements engineering and architecture design." IEE Proceedings - Software 151(4): pp. 153-155.

Rashid, A., P. Sawyer, A. Moreira and J. Araujo (2002). Early aspects: a model for aspect-oriented requirements engineering. Proceedings of the IEEE Joint International Conference on Requirements Engineering, Essen, Germany: pp. 199-202.

Reiff-Marganiec, S. (2004). Policies: Giving Users Control over Calls. Agents, Objects and Features. Berlin, Springer Verlag: pp. 189-208.

Reiff-Marganiec, S. and M. D. Ryan (2005). Feature Interactions in Telecommunications and Software Systems VIII. Amsterdam, The Netherlands, IOS Press.

Reiff-Marganiec, S. and M. D. Ryan (2007). "Guest Editorial." Journal of Computer Networks: Special Issue on Feature Interaction 51(2): pp. 357-358.

Reiff-Marganiec, S. and K. J. Turner (2004). "Feature Interaction in Policies." Computer Networks: The International Journal of Computer and Telecommunications Networking 45(5): pp. 569-584.

Robinson, W. N. and S. D. Pawlowski (1999). "Managing Requirements Inconsistency with *Development Goal Monitors*." IEEE Transactions on Software Engineering 25(6): pp. 816-835.

Robinson, W. N., S. D. Pawlowski and V. Volkov (2003). "Requirements Interaction Management." ACM Computing Surveys 35(2): pp. 132-190.

Ruhe, G. and M. O. Saliu (2005). "The art and science of software release planning." IEEE Software 22(6): pp. 47-53.

Russo, A., R. Miller, B. Nuseibeh and J. Kramer (2002). An Abductive Approach for Analysing Event-Based Requirements Specifications. Proceedings of the 18th International Conference on Logic Programming, Springer-Verlag: pp. 22-37.

Salifu, M., Y. Yu and B. Nuseibeh (2007). Specifying Monitoring and Switching Problems in Context. Proceedings of the 15th IEEE International Conference in Requirements Engineering (RE '07): pp. 211-220.

Scalera, S. M. and J. R. Vazquez (1998). The design and implementation of a context switching FPGA. Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines: pp. 78-85.

Sefidcon, A. and F. Khendek (2000). "FID: feature interaction detection tool." Microprocessors and Microsystems 24(6): pp. 283-289.

Shanahan, M. (1999). The Event Calculus Explained. Lecture Notes in Computer Science. Berlin / Heidelberg, Springer. 1600/1999: pp. 409.

Shaw, J., J. Giglierano and J. Kallis (1989). "Marketing complex technical products: The importance of intangible attributes." Industrial Marketing Management 18(1): pp. 45-53.

Shehata, M., A. Eberlein and A. Fapojuwo (2007a). "Using semi-formal methods for detecting interactions among smart homes policies." Science of Computer Programming 67(2-3): pp. 125-161.

Shehata, M., A. Eberlein and A. O. Fapojuwo (2007b). "A taxonomy for identifying requirement interactions in software systems." Journal of Computer Networks: Special Issue on Feature Interaction 51(2): pp. 398-425.

Siddiqi, S. and J. M. Atlee (2000a). "A hybrid model for specifying features and detecting interactions." Computer Networks 32(4): pp. 471-485.

Siddiqi, S. and J. M. Atlee (2000b). "A hybrid model for specifying features and detecting interactions." Journal of Computer Networks 32(4): pp. 471-485.

Silberschatz, A., P. B. Galvin and G. Gagne (2004). Operating System Concepts. London, John Wiley & Sons, Inc.

Sochos, P., I. Philippow and M. Riebisch (2004). Feature-Oriented Development of Software Product Lines: Mapping Feature Models to the Architecture. 5th Annual International Conference on Object-Oriented and Internet-based Technologies, Concepts, and Applications for a Networked World (Net.ObjectDays'2004). Messekongresszentrum Erfurt, Germany: pp.

Sousa, R. and C. A. Voss (2008). "*Contingency research in* operations management practices." Journal of Operations Management 26(6): pp. 697-713.

Spanoudakis, G. and K. Mahbub (2006). "Non Intrusive Monitoring of Service Based Systems." International Journal of Cooperative Information Systems 15(3): pp. 325-358.

Stafford, J. and K. Wallnau (2001). Predicting Feature Interaction in Component-Based Systems. 15th European Conference on Object-Oriented Programming, Hungary: pp.

Stepien, B. and L. Logrippo (1994). Representing and Verifying Intentions in Telephony Features Using Abstract Data Types. Third International Workshop on Feature Interactions in Telecommunications Software Systems, IOS Press: pp. 141-155.

Svahnberg, M., J. v. Gurp and J. Bosch (2005). "A taxonomy of variability realization techniques." Software: Practice and Experience 35(8): pp. 705-754.

Thiel, S., S. Ferber, T. Fischer, A. Hein and M. Schlick (2001). A Case Study in Applying a Product Line Approach for Car Periphery Supervision Systems. Proceedings of In-Vehicle Software 2001 (SP-1587), Detroit, Michigan, USA: pp. 43-55.

Tsang, S. and E. H. Magill (1997). Behaviour Based Run-Time Feature Interaction Detection and Resolution Approaches For Intelligent Networks. Feature Interactions in Telecommunication Networks IV. Amsterdam, IOS Press: pp. 254–270.

Tsang, S. and E. H. Magill (1998). "Learning To Detect and Avoid Run-Time Feature Interactions in Intelligent Networks." IEEE Transactions on Software Engineering 24(10): pp. 818-830.

Turner, C. R., A. Fuggetta, L. Lavazza and A. L. Wolf (1999). "A Conceptual basis for feature engineering." The Journal of Systems and Software 49(1): pp. 3-15.

Turner, K. J. (2000). Formalising the Chisel Feature Notation. Proceedings of the Feature Interactions in Telecommunications Networks VI. Amsterdam, IOS Press Amsterdam: pp. 241-256.

Turner, K. J. and L. Blair (2007). "Policies and conflicts in call control." Journal of Computer Networks: Special Issue on Feature Interaction 51(2): pp. 496-514.

Turner, K. J., E. H. Magill and D. J. Marples (2004). Service Provision, John Wiley & Sons, Ltd.

Uchitel, S. and M. Chechik (2004). Merging Partial Behavioural Models. ACM International Symposium on Foundations of Software Engineering (FSE'04), Newport Beach: pp.

Umanath, N. S. (2003). "The concept of contingency beyond "It depends": illustrations from IS research stream." Information & Management 40(6): pp. 551-562.

van Lamsweerde, A., R. Darimont and E. Letier (1998). "Managing conflicts in goal-driven requirements engineering." IEEE Transactions on Software Engineering 24(11): pp. 908-926.

van Lamsweerde, A. and L. Willemet (1998). "Inferring declarative requirements specifications from operational scenarios." IEEE Transactions on Software Engineering 24(12): pp. 1089-1114.

Velthuijsen, H. (1993). "Distributed artificial intelligence for runtime feature-interaction resolution." Computer 26(8): pp. 48-55.

Weiss, M. and B. Esfandiari (2004). On Feature Interactions Among Web Services. Proceedings. IEEE International Conference on Web Services.: pp. 88- 95.

Weiss, M., B. Esfandiari and Y. Luo (2007). "Towards a classification of web service feature interactions." Journal of Computer Networks: Special Issue on Feature Interaction 51(2): pp. 359-381.

Weston, N., R. Chitchyan and A. Rashid (2008). A Formal Approach to Semantic Composition of Aspect-Oriented Requirements. Proceedings of the 2008 16th IEEE International Requirements Engineering Conference, IEEE Computer Society: pp. 173-182.

Wilson, M., E. H. Magill and M. Kolberg (2005). An online approach for the service interaction problem in home automation. Proceedings of the 2nd IEEE Consumer Communications and Networking Conference: pp. 251-256.

Wong, K. C., J. G. Thistle, R. P. Malhameacute and H. H. H. (2000). "Supervisory Control of Distributed Systems: Conflict Resolution." Discrete Event Dynamic Systems 10(1-2): pp. 131-186.

Wu, X. and H. Schulzrinne (2007). "Handling feature interactions in the language for end system services." Journal of Computer Networks: Special Issue on Feature Interaction 51(2): pp. 515-535.

Xu, Y., L. Logrippo and J. Sincennes (2007). "Detecting feature interactions in CPL." Journal of Network and Computer Applications 30(2): pp. 775-799.

Yokogawa, T., T. Tsuchiya, M. Nakamura and T. Kikuno (2003). "Feature Interaction Detection by Bounded Model Checking." IEICE Transactions on Information and Systems 2003 E86-D(12): pp. 2579-2587.

Yoo, J., J. Catanio, R. Paul and M. Bieber (2004). "Relationship analysis in requirements engineering." Journal of Requirements Engineering 9(4): pp. 238 - 247.

Yu, P. S. and D. M. Dias (1993). "Performance analysis of concurrency control using locking with deferred blocking." Software Engineering, IEEE Transactions on 19(10): pp. 982-996.

Zave, P. (2001). Requirements for Evolving Systems: A Telecommunications Perspective. Fifth IEEE International Symposium on Requirements Engineering (RE '01), 2001, IEEE Computer Society: pp. 2-9.

Zave, P. and M. Jackson (1993). "Conjunction as composition." ACM Transactions on Software Engineering and Methodology (TOSEM) 2(4): pp. 379-411.

Zave, P. and M. Jackson (1997). "Four dark corners of requirements engineering." ACM Transactions on Software Engineering and Methodology (TOSEM) 6(1): pp. 1-30.

Zave, P. and M. Jackson (2002). A Call Abstraction for Component Coordination. Proceedings of the 29th International Colloquium on Automata, Languages, and Programming: Workshop on Formal Methods and Component Interaction, University of Malaga, Malaga, Spain: pp.

Zhang, J. and B. H. C. Cheng (2005). Specifying adaptation semantics. Proceedings of the 2005 workshop on Architecting dependable systems. St. Louis, Missouri, ACM Press: pp. 1-7.

Zhang, J., B. H. C. Cheng, Z. Yang and P. K. McKinley (2005a). Enabling Safe Dynamic Component-Based Software Adaptation. Architecting Dependable Systems III. Berlin / Heidelberg, Springer Berlin / Heidelberg. 3549 / 2005: pp. 194.

Zhang, W., H. Mei and H. Zhao (2005b). A feature-oriented approach to modeling requirements dependencies. Requirements Engineering, 2005. Proceedings. 13th IEEE International Conference on: pp. 273-282.

Zimmer, P. A. and J. M. Atlee (2005). Categorizing and Prioritizing Telephony Features. Feature Interactions in Telecommunications and Software Systems VIII. Leister, U.K., IOS Press: pp. 327-334.